密级_____

中国科学院大学
University of Chinese Academy of Sciences

博士学位论文

分组密码和 SHA-3 杂凑函数的差分分析

作者姓名: 乔珂欣

指导教师: 胡磊 研究员

中国科学院信息工程研究所

学位类别: 工学博士

学科专业: 信息安全

研究所: 中国科学院信息工程研究所

2017 年 5 月

Differential Cryptanalysis on Block Ciphers and SHA-3 Hash Functions

By
Ke-Xin Qiao

A Dissertation Submitted to
The University of Chinese Academy of Sciences
In partial fulfillment of the requirement
For the degree of
Doctor of Information Security

Institute of Information Engineering
Chinese Academy of Sciences

May, 2017

中国科学院信息工程研究所
研究生学位论文独创性声明

本人声明所呈交的学位论文是本人在导师胡磊指导下进行的研究工作及取得的研究成果。除了文中特别加以标注和致谢的内容外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得中国科学院信息工程研究所或其他教育机构的学位或证书而使用过的材料。与我共同工作的同志对本研究所做的任何贡献均已在论文中做了明确的说明并表示谢意。

论文与资料若有不实之处，本人承担一切相关责任。

学位论文作者签名：乔珂欣　　签字日期：2017年5月11日

学位论文版权使用授权说明

本学位论文作者完全了解中国科学院信息工程研究所有关保留、使用学位论文的规定。特授权中国科学院信息工程研究所可以将学位论文的全部或部分内容编入有关数据库进行检索，并采用影印、缩印或扫描等复制手段保存、汇编，以供查阅和借阅。同意研究所向国家有关部门或机构送交论文的复印件和磁盘。

(保密的论文在解密后应遵循此规定)

学位论文作者签名：乔珂欣　　导师签名：

签字日期：2017年5月11日　　签字日期：2017年5月11日

摘　要

分组密码和杂凑函数是信息安全的底层模块，在数据保密、身份认证、安全通信等方面起着重要作用。对分组密码和杂凑函数的设计及安全性分析和评估是现代密码学领域的重要研究内容。差分分析是对分组密码和杂凑函数最有效的分析方法之一。本文围绕差分分析进行了研究。杂凑函数方面，对 SHA-3 杂凑函数标准 KECCAK 系列运用代数分析与差分分析相结合的方法进行了约减轮碰撞攻击。分组密码方面，改进了基于线性规划的自动化差分搜索方法，并应用于 FOX 密码算法；将动态密钥猜测技术进行了自动化程序实现，并用于分析 SIMON 和 Simeck 算法。此外，本文研制了轻量级密码算法设计中常用的 4-比特 S 盒的高效硬件实现工具。具体研究工作和成果如下：

1. 给出了 SHA-3 杂凑函数标准 KECCAK 系列多个版本的 5 轮实际碰撞攻击，其中对 SHAKE128 版本的攻击结果是目前已知对 SHA-3 标准算法的最高轮数实际攻击。我们原创性地提出了可线性化仿射子空间的理念，用于将 SHA-3 底层置换函数中唯一的非线性部件 S 盒在子空间上进行线性化，从而将搜索满足 2 轮特定差分要求的消息对的问题转化为线性方程组的求解问题，方程组的解空间即提供了以概率 1 满足 2 轮差分要求的消息空间，进而在解空间中以实际可行的计算复杂度搜索后面 3 轮的消息碰撞，最终得到 5 轮碰撞。利用该方法，我们在少于 3 小时的单核计算时间内找到了 SHAKE128 和两个 KECCAK 挑战赛版本的 5 轮消息碰撞实例，并将前人 4 轮碰撞攻击计算复杂度的对数降低了一半。

2. 改进了基于线性规划的自动化差分搜索方法中对异或操作的建模，并得到对 Lai-Massey 结构分组密码 FOX 更准确的安全性估计。基于线性规划的自动化差分搜索方法将密码算法的高概率差分搜索问题转化为线性规划问题。我们发现该方法由于对差分传播模式描述上的粗糙性，无法得到 FOX 密码的有效差分迹。通过重构差分在异或操作上传播模式的线性不等式约束，改进后的方法完全消除了原方法应用于 FOX 密码时产生的无效差分迹。利用改进后的方法，我们得到了 FOX 密码算法的更紧的活

跃 S 盒个数下界，证明了 6 轮 (而非之前所认识的 8 轮) FOX64 的最高差分概率不超过 2^{-64}，足够抵抗基本差分攻击。

3. 将差分分析密钥恢复阶段的动态密钥猜测技术进行了自动化实现，并用于分析 SIMON 和 Simeck 分组密码算法。SIMON 算法是美国国家安全局提出的轻量级分组密码算法，Simeck 是 CHES 2015 上提出的类似 SIMON 的算法。动态密钥猜测技术是针对该类密码提出的，从攻击轮数上优于传统的密钥猜测方法，但是具有计算繁琐的缺点。我们将该技术进行了程序实现，程序可以自动输出密钥猜测过程中涉及的密钥比特和计算复杂度，大大减少了复杂度计算的工作量。结合本文找到的高概率差分和原有差分，我们给出了 21、22 轮 Simeck32，28 轮 Simeck48，34、35 轮 Simeck64 和 22 轮 SIMON32/64，24 轮 SIMON48/96，28、29 轮 SIMON64/96，29、30 轮 SIMON64/128 的差分分析结果。

4. 研制了 4-比特 S 盒的低电路深度实现工具。用 NOT、AND、NAND、OR、NOR、XOR、XNOR 逻辑门实现 S 盒时，低电路深度意味着低延迟和高时钟频率。将 S 盒的每个输出比特看作输入比特的布尔函数，用递归算法枚举出一定深度阈值内可以实现的布尔函数，将真值表和最低延迟的实现方式存储下来。对任意给定 4-比特 S 盒，解析出其对应的布尔函数真值表，即可即时给出深度最低的硬件实现方式。利用该工具，我们还给出了具有最优差分性质和线性性质且能在最小深度内实现的 4-比特 S 盒。

关键词： 分组密码；杂凑函数；SHA-3 算法；差分分析；线性规划；线性化技术；碰撞攻击；S 盒

Abstract

Block ciphers and hash functions are fundamental components of information security, essential to data confidentiality, identity authentication and secure communications, *etc.* Design, cryptanalysis and security evaluation of block ciphers and hash functions are important studies in modern cryptology. In this dissertation, we study differential cryptanalysis, which is one of the most effective cryptanalysis methods on both ciphers. For hash functions, we present practical 5 round collision attacks on SHA-3 hash function family, namely KECCAK, by a developed combinational method of algebraic and differential cryptanalysis. For block ciphers, firstly we improve a linear programming based automatic method for differential search and apply to FOX. Secondly we develop a program for dynamic key-guessing techniques and apply to SIMON and Simeck. Besides, we study the hardware implementation of 4-bit Sboxes with least circuit depth. Our works and innovations are as follows.

1. We present practical 5 round collision attacks on several versions of SHA-3 hash function family KECCAK, among which is a cryptanalysis on SHAKE128 covering the most rounds ever known in practical attacks. We originally propose an idea and a concept of linearizable affine subspaces and linearize the first Sbox layer in SHA-3 permutation function. Thus we convert the problem of finding message pairs satisfying 2-round differentials to that of solving a linear equation system. The solution space of the system provides message pairs satisfying the first 2-round differentials with probability 1. Searching for collisions covering the latter 3 rounds is conducted in the solution space. Finally, we get 5 round collision messages of SHAKE128 and two KECCAK challenge versions. The complexity of previous 4 round collision attacks is also decreased.

2. We improve modeling method for XOR arithmetic in linear programming based automatic method for differential searching, and achieve more accu-

rate security evaluation of FOX, a cipher in Lai-Massey structure. In linear programming based automatic method, finding differentials with high probability is converted to solving a linear programming problem. We discover that due to inaccuracy of descriptions for differential propagation, this method doesn't work for FOX to get effective differentials. By refining the linear inequations describing the differential propagation on XOR, we eliminate the illegal differentials that appear in previous method applying to FOX. With the developed method, we get tighter lower bounds for number of active Sboxes in FOX, and prove that 6 (not previously known 8) round FOX64 is strong enough to resist against basic differential attacks as the differential probability is bounded by 2^{-64} .

3. We implement dynamic key-guessing techniques in differential cryptanalysis by program and apply to SIMON and Simeck. SIMON is a family of lightweight block ciphers proposed by National Security Agency. Simeck is proposed in CHES 2015 which is very similar to SIMON. Dynamic key-guessing techniques are aimed at such ciphers to improve the number of rounds attacked by differential method. However, this kind of method bears complicated analyzing procedure. Therefore, we implement this method to an automatic program, which can automatically output the complexities and relevant data during the key guessing procedure. We launch cryptanalysis on 21, 22-round Simeck32, 28-round Simeck48, 34, 35-round Simeck64 and 22-round SIMON32/64, 24-round SIMON48/96, 28, 29-round SIMON64/96 and 29, 30-round SIMON64/128.

4. We develop a tool for hardware implementation of 4-bit Sboxes with least circuit depth by logical gates NOT, AND, NAND, OR, NOR, XOR, XNOR. Lower circuit depth means shorter delay and higher clock frequency. Taking each output bit of a given Sbox as a boolean function of input bits, we enumerate all boolean functions that can be implemented under a certain depth threshold, and record their truth tables and implementation. For any given Sbox, this tool can output its implementation instantly by extracting its output boolean functions. We also use this tool to suggest 4-bit Sboxes

with best differential and linear property that can be implemented in least depth simultaneously.

Keywords: block cipher, hash functions, SHA-3, differential cryptanalysis, linear program, linearization techniques, collision attack, Sbox

目　录

表　格

插　图

第一章　引言

1.1　背景和意义

密码学作为信息安全的重要基础，在信息化时代经历了蓬勃的发展，随着未来人工智能时代的来临、物联网的发展，具有数据保密、身份认证、安全通信等功能的硬件设施将越来越普遍地应用在生产生活中，这些功能都需要依赖密码学实现。密码学的重要性在未来有增无减。现代密码学包含了对称密码、公钥密码、安全协议等多方面的内容，其中分组密码和杂凑函数是对称密码学的重要研究对象，在数据加密、数据完整性认证、身份认证、随机数生成等方面起重要作用，是信息安全的基础模块。分组密码是将特定长度 (如 256 比特、128 比特、64 比特等) 的数据块进行加密的算法，有别于流密码对数据流逐比特或逐字节的处理，具有加密效率高、易于标准化等优点。杂凑函数将任意长度的数据转化成一个特定长度的摘要值，生成的摘要需要具有数据“指纹”的功能。

现代分组密码通常采用迭代型设计，即通过将一定结构的轮函数进行多次迭代来充分实现数据的扩散和混淆。最常见的两种结构是 Feistel 结构和 SPN 结构。Feistel 结构的特点是每轮轮函数处理一半分组的数据，然后交换两半，这种做法使得加解密过程相同，在算法实现上有优势。采用 Feistel 结构的代表算法如 1977 年被美国国家标准和技术协会 (National Institute of Standards and Technology, NIST) 采纳为 FIPS PUB 46 标准的 DES 算法 [97]，被 ISO/IEC 29192-2 采纳为轻量级分组密码标准的 CLEFIA 算法 [113]，由美国国家安全局 (National Security Agency, NSA) 于 2013 年提出的面向硬件和软件实现的 SIMON 和 SPECK 算法 [13]，以及 HIGHT [61]、MIBS [65]、LBlock [134]、Piccolo [112]、TWINE [120]、Simeck [137] 等等。SPN 结构的特点是每轮轮函数采用替换 (substitution) 和置换 (permutation) 两层操作，代表算法如 NIST 于 2001 年发布的高级加密标准 AES [98]，同样被 ISO/IEC 29192-2 采纳为轻量级分组密码标准的 PRESENT 算法 [30]，以及 Serpent [22]、ICEBERG [114]、PRINTcipher [73]、KLEIN [55]、PRINCE [31]、RECTANGLE [140]、Midori [12]、SKINNY [14] 等等。另外，1991 年由来学嘉和 Massey 提出的 Lai-Massey 结构 (LM 结构) [78, 79] 也

是一种分组密码设计结构，后来由 Vaudenay 进行了改进 [123]。

杂凑函数的概念最早出现在 Diffie 和 Hellman 发表在 1976 年开辟了密码学新方向的论文中 [44]，在数字签名、口令验证、密钥产生等现代网络安全的各个领域都发挥着作用。杂凑函数需要满足的两个最基本的安全性要求是单向性和抗碰撞性。起初最为经典的设计是 1989 年产生的 Merkle-Damgård 结构 (MD 结构) [42, 90]，1990 年和 1991 年由 Rivest 提出的 MD4 [109] 和 MD5 [108] 杂凑函数都是基于这一结构的设计。NIST 在 1993 年、1995 年和 2001 年相继推出的杂凑函数标准 SHA-0、SHA-1 和 SHA-2 算法 [99] 都采用了与 MD4/5 类似的设计。2005 年，王小云的团队在这一系列杂凑函数算法的攻击上做出了突破性的工作，给出了 MD4、MD5、SHA-0 等算法的实际碰撞结果 [127, 130, 131]，以及 SHA-1 算法碰撞攻击的理论结果 [128]。由于 SHA-2 算法采用了类似的结构，业界对其安全性产生忧虑，于是 NIST 自 2006 年开始筹备 SHA-3 竞赛，旨在选出杂凑函数标准的补充算法。2007 年 11 月，NIST 正式公布 SHA-3 竞赛，一年后确定了 51 个参赛算法，经过两轮的评定和筛选，2012 年 10 月 NIST 公布由 Bertoni 等提出的 Keccak 算法胜出，将被推选为 SHA-3 标准。2014 至 2015 年 NIST 推出 FIPS PUB 202 标准，SHA-3 标准正式确立。SHA-3 采用了 Sponge 结构 (中文意为“海绵”，多数情况下我们仍用 Sponge 这个名称)，该结构也逐渐成为密码设计的热门结构，在 2013 年同样由 NIST 发起的认证加密算法竞赛 (CEASER 竞赛) [96] 中被多个候选算法所采用，如通过三轮评选的候选算法 Ketje [16]、Keyak [17]、Ascon [49]、NORX [9]，通过两轮评选的算法 ICEPOLE、STRIBOB、Pi-Cipher、PRIMATEs 等 [15]。

与分组密码和杂凑函数的产生和设计相伴而成的即是对它们的安全性分析。对分组密码的分析主要从恢复密钥的角度进行，对杂凑函数的分析主要从原像攻击和碰撞攻击的角度进行。对杂凑函数而言，不管是 MD 结构还是 Sponge 结构，底层模块通常是类似分组密码的置换，依然使用迭代型设计，所以对杂凑函数和分组密码的分析具有共通性。使用最为广泛且有效的分析方法是由 Biham 和 Shamir 在 1991 年提出的差分分析 [25]，之后不断发展出截断差分攻击 [74]、高阶差分攻击 [74]、不可能差分攻击 [23]、飞来去器攻击 [124]、矩形攻击 [24]、反弹攻击 [88] 等多种基于差分的攻击方法。其他分析方法，如代数方法、线性分析方法 [85] 等也用于与差分分析方法相结合，产生了差分-代数分析 [3]、差分-线性分析 [80] 等分析方法，可提高对某些密码的分析能力。差分分析还被用于可获取密码设备与外界物理交互信息的侧信道 [70] 攻

击方法中，产生了差分差错注入 [26]、差分能量分析 [75] 等分析方法。几乎所有对称密码算法在提出后都会受差分分析方法及相关方法的攻击和安全性评估，这使得密码设计者在设计新的密码算法时务必考虑其抗差分系列攻击的安全性。

对分组密码和杂凑函数的差分分析是密码学领域经典且具有重要实际意义的研究，在分析方法上的改进、对密码算法新的攻击结果都会推动对称密码学领域的发展，带动新的思考。

对密码算法而言，除了考虑其安全性，还要考虑其实现效率。目前对适用于小型资源受限设备上的轻量级密码算法的需求越来越高，算法能耗、时间延迟、门电路面积、吞吐量等指标用于衡量密码算法满足轻量级需求的能力，密码设计者们尽可能地探索密码算法的轻量级实现，即能耗更小、延迟更低、门电路面积更小、吞吐量更高的实现方式。对密码算法的轻量级实现基于对每个密码学部件的轻量级实现，包括线性层、S 盒等的轻量级实现。

1.2 国内外发展现状

差分分析是为攻击分组密码而提出的，基本差分分析的关键技术是搜索高概率差分区分器，即某一个输入差分在经过若干轮加密轮函数后以能区别加密函数和随机置换的高概率得到某一输出差分。有了差分区分器，就可以在区分器两端添加若干轮进行分组密码的恢复密钥攻击。Matsui 于 1994 年提出的分支定界算法 [86] 是搜索高概率差分的经典方法，提出时针对的是 Feistel 结构。但是在有些算法如 FEAL [91] 上时间复杂度太高，并不适用。之后针对 FEAL 算法出现了分支定界搜索方法的改进 [7]。同时，这个方法也被扩展到 SPN 结构上 [39]。分支定界算法及其改进算法的一个显著缺点是针对不同的目标密码算法需要专门编写程序，代码量大，可移植性差，难以形成系统的方法推广使用。而对于含 S 盒的密码算法，基于整数规划的自动化方法可以很好地避免这些缺点。

对于含 S 盒的密码，可以通过整体最小化活跃 S 盒个数获得对高概率差分的估计，从这个角度出发产生的操作简单、易实现、可移植性强的自动化方法得到密码分析者的广泛关注。2009 年，Borghoff 等 [32] 将流密码 Bivium [43] 加密过程中的二次方程转化为线性规划问题，然后求解得到的规划问题来恢复算法的初始状态。2012 年，Mouha 等 [93] 将基于整数规划的自动化方法用于

求最小活跃 S 盒个数，通过用线性不等式来刻画差分在密码算法上的传播，将目标函数设置为活跃 S 盒个数，从而将高概率差分的估计转化成线性规划问题。在分组密码 AES 上的应用结果与基于宽轨道策略 [40] 证明得到的结果完全一致，极好地肯定了这种方法的合理性。2013 年，Sun 等 [117] 将 Mouha 等的方法扩展到比特级密码算法上，并且成功证明了全轮的 PRESENT-80 可以抵抗单密钥下的差分攻击，并且给出了约减轮抵抗相关密钥差分攻击的安全性估计。此后对 Sun 等方法中个别部件的建模方法又有了进一步的改进 [105]。基于整数规划的自动化搜索方法在字节级算法上表现较好，但推广到比特级算法上时，因为变量个数、约束个数增多，S 盒刻画复杂使得计算量太大。2014 年，Sun 等对轻量级 S 盒可行差分传播模式形成的凸闭包进行精确刻画 [119]，剪切掉对 S 盒的冗余约束，只保留最有效的约束，从而简化线性规划问题的约束条件。利用改进后的方法，他们给出了全轮 PRESENT-80 在相关密钥模型下差分迹概率的更紧的上界，以及全轮 LBlock 密码算法在相关密钥模型下最优差分迹概率的上界，在单密钥、相关密钥差分搜索上都有新的提升。2016 年，基于线性规划的自动化差分搜索方法被推广到由模加运算、循环移位、异或运算构成的 ARX 类型的密码上，并在 NSA 提出的 Speck 算法上得到很好的应用 [52]。

自动化差分搜索方法在各种变种差分分析方法中也有很好的应用。截断差分攻击所需要的截断差分只需要确定一部分位置上的差分值，对另外一些位置的差分值不做要求。飞来去器攻击方法和矩形攻击方法需要搜索两段高概率差分，然后交叉拼接形成四元组差分区分器。基于整数规划的自动化搜索方法只需减少或增加一些约束条件，都可以满足这些特定差分的搜索需求 [104]。对于不可能差分，基于整数规划的方法最直接的应用是验证一个输入、输出差分对是否构成不可能差分，方法是验证得到的线性规划问题是否无解；在不可能差分的搜索上可以通过逐一排查某些形式的输入、输出差分对是否构成不可能差分 [138] 来进行。不可能差分的搜索有另外一系列自动化方法，如 2003 年提出的 U-方法 [72]，用矩阵描述密码算法，中间相遇产生矛盾得到不可能差分；2009 年产生的 UID-方法 [84] 去除了 U-方法中的一些限制，利用了更多产生矛盾的条件；2012 年提出的截断 IDC 自动化搜索方法 [133] 从算法整体寻找矛盾，而不只局限于中间相遇产生的矛盾，在 AES，Camellia [6] 等算法上能找出更多的不可能差分。

与之前计算 SPN 结构 [40]、Feistel 结构和广义 Feistel 结构 [28, 29, 69,

111] 的对称密码算法活跃 S 盒个数下界的方法相比，基于整数规划方法的优势在于代码量更小，且能够自动化地估计密码算法在单密钥和相关密钥模型下的安全界。这种方法只需要通过一个简单的程序描述对称密码算法的结构来产生整数规划问题算例，该算例需要进行最小化的目标函数就是差分或线性活跃 S 盒的数量。用一个第三方的优化工具就可以求解这个算例。基于整数规划的技术可以广泛应用于由 S 盒、线性置换和异或运算构成的密码算法上。

配合不同的分析对象，与差分分析相结合的各种适应性方法也不断发展。2009 年，代数方法与差分方法相结合的方法首次被提出 [3]，分析了 PRESENT-128 密码算法，并于 2011 年得到进一步的发展 [125]。2014 年，动态密钥猜测技术 [126] 被用于分组密码差分分析的恢复密钥阶段，该方法将经典的差分分析 [25] 和广泛应用于杂凑函数分析的模差分分析 [36, 82, 87, 121, 129] 相结合，主要针对以按位与作为非线性操作的密码算法。

尽管差分分析是为了攻击分组密码而提出的，但是在杂凑函数的分析上也得到了极好的应用，且成果显著，其中最为著名的是 2005 年王小云教授对 MD 系列算法的分析结果，包括对 MD4/5、SHA-0 等著名算法的实际碰撞攻击 [127, 130, 131] 和对 SHA-1 算法的理论攻击 [128]。这一系列攻击方法巧妙结合了异或差分与模加差分，用启发式的方法改进了所需差分的搜索。由于 SHA-2 算法与被实际攻破的算法有相似的设计，如在消息链接上同样采用了 MD 结构，内部置换也同样采用了 ARX 结构，王小云教授的一系列实际攻击结果使业界对 SHA-2 算法安全性的信心受到冲击。2007 年 11 月，NIST 正式公布了 SHA-3 竞赛，面向全球征集新一代杂凑函数标准，即 SHA-3 标准。2008 年 12 月收到了 51 个参赛算法，2009 年 7 月，14 个算法进入第二轮评审，2010 年 12 月，五个算法进入最终的角逐，这五个算法是：BLAKE [8]、Grøstl [54]、JH [132]、Keccak [20]、Skein [51]。对这些候选算法的分析促进了对杂凑函数的分析方法的发展，如内部差分分析 [102]、多比特广义差分攻击 [81, 82]、反弹攻击 [89]、中间相遇攻击 [71] 等。2012 年 10 月，NIST 宣布 Keccak 算法胜出，SHA-3 竞赛结束。2015 年 8 月，NIST 发布的 SHA-3 标准文档 FIPS PUB 202 正式确立，包含了 Keccak 系列算法中的六个算法：Keccak-224、Keccak-256、Keccak-384、Keccak-512、SHAKE128、SHAKE256。Keccak 的设计者为了使算法的安全性得到充分论证，于 2009 年发起了攻击比赛，后来发展为截止日期无限开放的 Keccak 碰撞攻击和原像攻击密码挑战赛 [18]。挑战赛中包含了更多状态缩小的版本供挑战者分析。2017 年 2 月 23 日，荷兰 CWI 研究所和

Google 公司的研究人员发布了首个 SHA-1 碰撞攻击实例 [115]，标志着长期以来从理论上对 SHA-1 算法安全性的警示终于落实成实际的攻击，这使得对新一代杂凑函数算法的研究需求更加迫切。

KECCAK 系列算法作为 SHA-3 竞赛的优胜算法受到了密码分析者的广泛关注。2009 年，Aumasson 等 [10] 实际构造了 9 轮 KECCAK 置换的零和区分器和达 15 轮的理论结果 (通常低于生日界的分析结果才被认为是有效的，而有部分结果虽然攻击轮数很高，但是实现复杂度已高于生日界 [33, 34])。2011 年，对 2 轮的 KECCAK-224 和 KECCAK-256 的碰撞攻击和第二原像攻击结果被发现 [95]。2012 年，Dinur 等 [45] 将碰撞攻击轮数提高到 4 轮，采用了代数方法与差分方法相结合的分析方法，给出了 KECCAK-224 和 KECCAK-256 的 4 轮实际碰撞攻击，以及 5 轮的近似碰撞攻击。对 KECCAK 内部置换函数差分传播性质的研究 [41, 50] 为差分分析提供了便利。2013 年，Dinur 等 [46] 又给出了 3 轮 KECCAK-512 和 KECCAK-384 的实际碰撞攻击，以及 5 轮 KECCAK-256 和 4 轮 KECCAK-384 的理论结果，采用了扩展的内部差分分析方法 [102]。2014 年，对 7-9 轮原像攻击的理论结果给出 [38]，但攻击复杂度仍大大超出目前的计算能力。2015 年，对密钥模式的 KECCAK 的立方攻击 [48] 可以达到 9 轮，其中对 6 轮的攻击复杂度降低到实际计算能力可达到的复杂度。2015 年，6-8 轮的区分器也被以更低的复杂度构造出来 [67]。2016 年，3 轮的实际原像攻击结果及 4 轮的理论结果被给出 [58]，并且 Aumasson 等构造的零和区分器覆盖的轮数在不提高复杂度的情况下被提高了 2 轮。

在 S 盒的低延迟实现上，文献 [101] 中给出了 Serpent 算法的 8个 4 比特 S 盒及其逆置换在 x86 处理器上的实现，使用了更少的指令序列和更少的寄存器。文献 [122] 对所有 4×4 比特 S 盒按照差分性质和线性性质进行分类，搜索出每个等价类中实现效率最高的 S 盒。文献 [37] 给出了 AES 的 8 比特 S 盒采用更少门电路的实现方式。文献 [35] 中探究了布尔函数最小化问题的求解复杂度。文献 [116] 利用可满足性问题 (SAT) 求解器给出了一系列算法尤其是 CAESAR 竞赛候选算法中的 S 盒实现，将非线性指令复杂度、位片门复杂度、门复杂度、电路深度复杂度等指标作为优化实现的目标。

1.3 本文工作

在对上述杂凑函数和分组密码的差分分析方法进行了调研和学习后，本文

对部分方法进行了改进，并应用在一系列密码算法上，取得了新的结果。本文第二章给出了进行相关研究所需的预备知识，本文工作及后续章节内容如下所述。

第三章给出了 SHA-3 杂凑函数标准 KECCAK 系列部分算法的 5 轮实际碰撞攻击。Dinur 等发表在 FSE 2012 的 4 轮碰撞攻击结果采用了将 1-轮连接器和 3-轮差分迹相结合的方法，在他们的这项工作基础之上，我们将连接器向前扩展了一轮，从而获得了 5 轮碰撞攻击结果。扩展方法得益于 KECCAK 杂凑函数较大的内部状态。对于 KECCAK 的 S 盒，当其输入限制在一个子空间上时，S 盒变换即等价于一个线性变换，从而可以在整个状态上线性化 S 盒层。通过线性化 KECCAK 置换函数第一轮的 S 盒层，求解满足 2-轮连接器要求的消息对的问题转化成了求解线性方程组的问题。线性化过程使得状态空间的自由度下降，但是只要连接器解空间的大小足够满足 3-轮差分迹所需数据量要求，即可在前 2 轮连接器的解空间中进行后 3 轮消息碰撞的搜索，找到 5 轮的消息碰撞。通过实验，我们实际得到了 SHA-3 标准之一 SHAKE128 和两个 KECCAK 挑战赛版本的 5 轮消息碰撞实例。这是国际上首个对 5 轮 SHA-3 标准系列碰撞攻击的实际结果。此约减轮的碰撞攻击结果并不影响全轮算法的安全性。

第四章给出了改进的基于整数规划的自动化差分搜索方法及其对 LM 型分组密码 FOX 的应用结果。计算差分活跃 S 盒个数在评估分组密码抗差分攻击安全性上十分重要，Mouha 等提出了基于混合整数线性规划 (Mixed-Integer Linear Programming, MILP) 技术来自动化求解活跃 S 盒个数下界的方法，Sun 等通过引入 S 盒的比特级表示将这一方法推广到比特级算法上。这类方法的有效性极大地依赖于线性不等式对差分传播方式描述的精确程度，更精确的描述才能得到更紧的活跃 S 盒个数下界，从而得到对密码算法安全性更精确的估计。我们将 Sun 等方法中的差分在异或运算上传播模式的描述方法进行了改进，并运用在 FOX 密码算法上。实验结果表明，改进后的方法消除了原有方法在 FOX 密码上产生的无效差分迹，得到 6 轮 FOX64 差分迹的概率已经低至 2^{-64}，与随机置换相同，从而证明 6 轮而非之前所认为的 8 轮 FOX64 即可抵抗基本单密钥差分攻击。我们也得到了 5 轮 FOX128 差分活跃 S 盒个数的更紧的下界，同样可以证明全轮 FOX128 可以抵抗单密钥差分攻击。此结果是对 FOX 密码算法安全性的更准确评估。

第五章对 SIMON 和 Simeck 分组密码算法进行了差分分析，在密钥恢复阶

段运用了自动化的动态密钥猜测分析。CHES 2015 上提出的 Simeck 算法结合了 NSA 提出的轻量级密码算法 SIMON 和 SPECK 的设计。动态密钥猜测技术于 2014 年针对 SIMON 算法提出。我们将动态密钥猜测技术进行了程序实现，程序可以自动化地给出密钥猜测阶段的数据，从而使得对 SIMON 和 Simeck 类型分组密码的安全性评估更方便地进行。我们用基于 MILP 的方法搜索了一条 Simeck 的高概率差分迹，并利用前人给出的差分迹，结合自动化的动态密钥猜测技术进行了 21、22 轮 Simeck32，28 轮 Simeck48 和 34、35 轮 Simeck64 的差分分析。利用 CRYPTO 2015 上发表的 SIMON 算法的高概率差分迹，进行了 22 轮 SIMON32/64，24 轮 SIMON48/96，28、29 轮 SIMON64/96 和 29、30 轮 SIMON64/128 的差分分析。对 SIMON64 的分析是目前轮数最多的分析结果。

第六章开发了 4×4 比特 S 盒的低延迟评估和实现工具。将每个输出比特看作输入比特的布尔函数，当以 AND、OR、XOR、NOT、NAND、NOR、XNOR 为操作指令时，用递归算法枚举出一定延时内可以实现的布尔函数，并将真值表和最低延迟的实现方式存储下来。当给定一个 4 比特 S 盒时，可解析出其对应的布尔函数真值表，通过与存储下的布尔函数的比对即时得到最低延迟的实现方式。我们还利用该工具给出了低延迟下可实现且具有最优差分、线性性质的优选 S 盒，可以用于轻量级密码算法的设计。

第七章对本文内容进行总结，并探讨了需要进一步研究的问题。

第二章　预备知识

2.1　分组密码

分组密码将需要加密的消息分成固定长度的消息块，然后对每个消息分块在密钥的参与下按照一定的算法进行加密。需要加密的消息称为明文，加密后的消息称为密文[1]。设分组长度为 b 比特，密钥长度为 k 比特，则加密过程可以表示为

$$E : \mathbb{F}_2^b \times \mathbb{F}_2^k \to \mathbb{F}_2^b,$$
$$(P, K) \mapsto C,$$

其中 P 为明文，K 为密钥，C 为密文。

分组密码一般采用迭代的形式，即将某个函数进行多次复合，使得扩散性和混淆性加强，从而提高整个密码算法的安全性。需要被多次复合的函数称为轮函数，复合的次数即为轮数。参与轮函数操作的密钥称为轮密钥或子密钥，由主密钥经过一定的密钥编排算法得来。图 2.1 是迭代型分组密码的加密过程。根据轮函数结构的不同，比较常见的是 Feistel 结构 (见图 2.2) 和 SPN 结构 (见图 2.3)。Feistel 结构的轮函数先处理分组的一半，然后交换两半。这一设计使得加解密算法相同，在算法实现上可以节省资源。SPN 结构的轮函数由一层非线性替换层和一层线性变换层构成。若分组密码是以字节作为最小分支进行加密处理的，可称为字节级密码，如 AES 就是典型的字节级密码。相应的也可以有半字节级密码，如 Midori 密码 [12]；比特级密码，如采用了比特拉线的 PRESENT 密码 [30]。为方便描述，后文将最小分支的规模在半字节及以上的密码统称为字级密码，以区分比特级密码。

分组密码的算法设计是公开的，需要保密的是密钥，密钥决定了明文分组被映射成何种密文分组。对分组密码的攻击也表现为以尽可能低的计算复杂度恢复密钥。根据攻击者掌握信息能力的不同，对分组密码的分析可以有唯密文攻击、已知明文攻击、选择明文攻击、选择密文攻击等。

[1]分组密码通常在一定的工作模式下使用，当我们只关注分组密码本身而不关注工作模式时，通常把分组密码算法的输入称作明文，算法输出称作密文，它们可能并不是原始的消息分块和最终的密文分块。

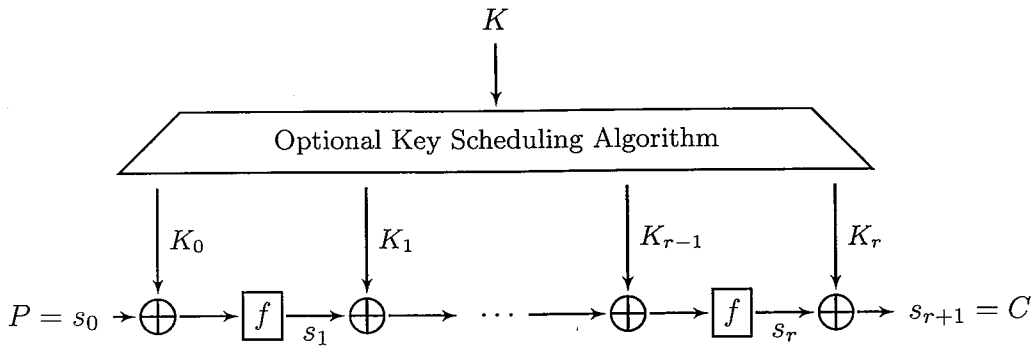


图 2.1: 迭代型分组密码 [66]

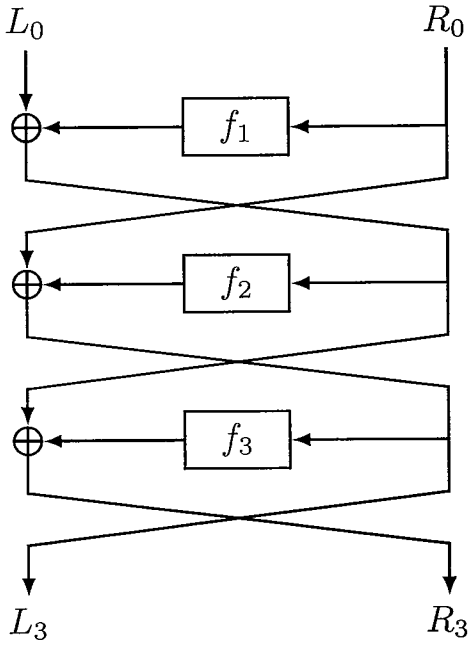


图 2.2: Feistel 结构 [66]

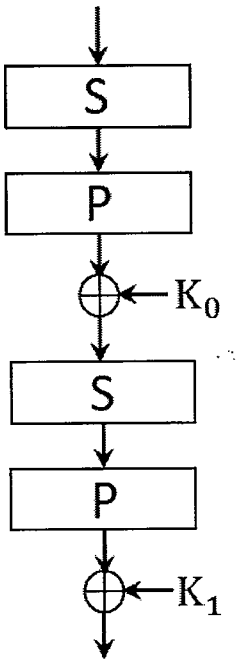


图 2.3: SPN 结构

唯密文攻击 攻击者仅知道一些密文。

已知明文攻击 攻击者知道用待破解密钥加密的一些明文和对应的密文。

选择明文攻击 攻击者可以选择明文，获得用待破解密钥进行加密的密文。

选择密文攻击 攻击者可以选择密文，获得用待破解密钥进行解密的明文。

对分组密码最直接的攻击方法是密钥穷搜，若密钥长度为 k，那么对 2^k 个密钥进行穷搜并通过将密文恢复为明文进行有效验证即可，因此只有复杂度低于 2^k 的攻击才被视为有效的攻击。同时，当分组长度 b 小于密钥长度 k 时，通过穷尽 2^b 明文分组即可知道所对应的密文，即获取了整个密码本，因此有效的攻击复杂度也应该小于 2^b。对迭代型分组密码而言，随着轮数的增加，攻击难度也会增大，因此设计较好的分组密码通常只存在约减轮的攻击，攻击轮数也被攻击者视为目标之一。

2.2　杂凑函数

密码学中的杂凑函数是一个将任意长度的输入消息处理为一个固定长度的短消息的函数，若输出消息长度为 d，杂凑函数可表示为：

$$\begin{aligned} h : \mathbb{F}_2^* &\to \mathbb{F}_2^d \\ x &\mapsto y, \end{aligned}$$

其中 x 是任意长度的消息，y 是长度固定为 d 的短消息，通常称为消息摘要或杂凑值。

杂凑函数需要满足的基本密码学性质有单向性和抗碰撞性。

单向性 给定消息摘要 y，找到消息 x 使得 $y = h(x)$ 是困难的。

抗碰撞性 找到两个不同的消息 x 和 x' 满足 $h(x) = h(x')$ 是困难的。

杂凑函数的安全性强度由输出摘要的长度决定。设杂凑函数的输出摘要长度为 n 比特，则单向性的理想安全性强度为 2^n，即找到一个消息摘要的原像平均需要复杂度为 2^n 的计算；抗碰撞性的理想安全性强度为 $2^{n/2}$，即找到一对消息碰撞平均需要复杂度为 $2^{n/2}$ 的计算，这一安全界是由生日攻击 [64] 所决定的，也称为生日界。任何所需计算复杂度小于理想安全性强度界的攻击都可以视为有效攻击。

为了处理任意长度的消息，杂凑函数通常将消息分块，然后迭代进行处理，消息块之间用一定的链接结构联系起来。最经典的结构设计有如图 2.4 所示的 MD 结构和图 2.5 所示的 Sponge 结构。MD 结构中消息进行填充后分成等长的消息块，然后迭代地用压缩函数 f 来处理消息分块，f 的输入为当前消息

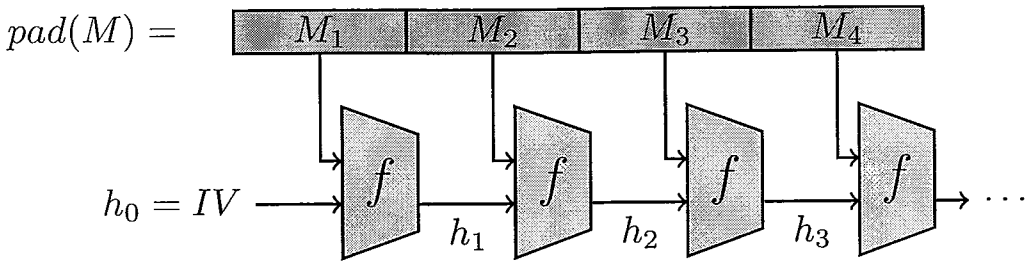


图 2.4: MD 结构 [66]

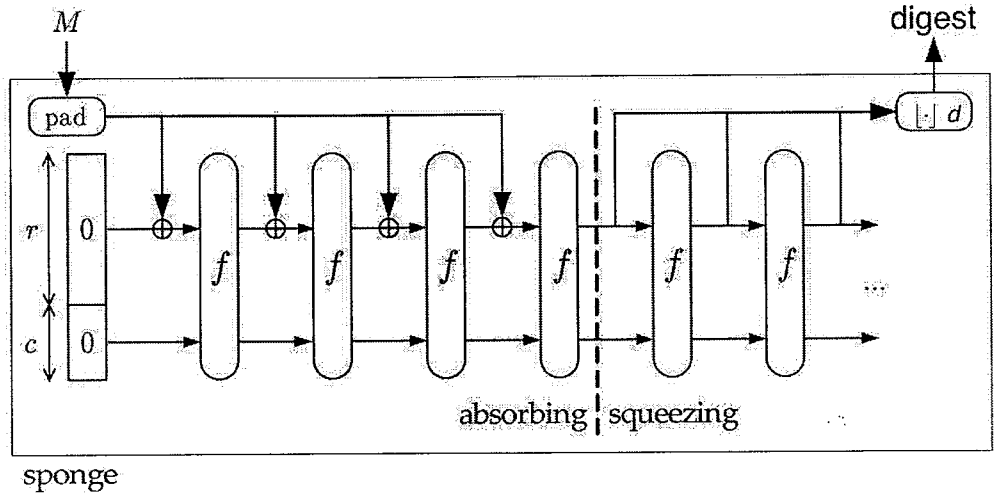


图 2.5: Sponge 结构 [19]

分块和上一次迭代产生的链接值，输出为用于下一次迭代的链接值。初始向量 IV 作为第一次迭代的链接值。Sponge 结构分为吸收阶段和挤出阶段。同样在进行消息填充后，将消息分成等长的 r 比特消息块。在吸收阶段，初始状态为全 0 比特，消息分块异或到状态的前 r 比特，然后整个状态进行置换操作 f，重复这一过程直到所有消息分块都被吸收并处理。在挤出阶段，将状态的前 r 比特输出，对整个状态进行置换函数 f 的操作，重复这一过程直到获得需要的摘要长度。

MD 结构和 Sponge 结构对每个分块的处理都是通过一个压缩函数或置换函数进行的。压缩函数和置换函数的设计通常也是迭代型的，因此对分组密码和杂凑函数的分析有相同之处，下面介绍的差分分析对二者都适用。

2.3　差分分析

2.3.1　差分分析概述

差分分析 [25] 是对迭代型密码最有效的分析手段之一。以分组长度为 n 的分组密码为例，在某密钥 k 下，令 $y = E(x)$ 表示加密函数 $E(\cdot)$ 对明文分组 x 的加密，y 是得到的密文分组。我们有如下基本概念：

差分值： $\Delta x = x \oplus x'$ 称为 x 和 x' 的差分值。尽管可以定义多种运算下的差分，本文仅考虑异或差分。

差分概率: 对于定义在 $\mathbb{F}_2^n$ 上的函数 F，输入差分 Δ_{in}，输出差分 Δ_{out}，概率

$$\begin{aligned} & Pr_x[F(x) \oplus F(x \oplus \Delta_{in}) = \Delta_{out}] \\ =& 2^{-n} \cdot \#\{x : F(x) \oplus F(x \oplus \Delta_{in}) = \Delta_{out}\} \\ =& 2^{-w} \end{aligned}$$

为从 Δ_{in} 到 Δ_{out} 的差分概率。若概率为 2^{-w}，则 w 称为差分的重量。

差分迹: 将每经过一定操作后的差分连接起来，形成一条差分迹。如 R 轮迭代型密码中，将每轮差分连接起来形成的差分迹可表示成

$$\Delta_0 \to \Delta_1 \to \Delta_2 \cdots \to \Delta_R.$$

差分闭包: 具有相同输入差分值和相同输出差分值的差分迹共同构成所覆盖函数的差分闭包。在无歧义的情况下，也将差分迹或差分闭包简称为差分。

活跃 S 盒: 输入差分值非零的 S 盒。

对于一个 $\mathbb{F}_2^n$ 上的随机函数 F 而言，对任意给定的非零输入差分值 Δ_{in} 和输出差分值 Δ_{out}，差分概率满足 $Pr_x[F(x) \oplus F(x \oplus \Delta_{in}) = \Delta_{out}] = 2^{-n}$，即输入差分值导致的输出差分值均匀分布在取值空间上。对于加密算法，若某输入差分值在一定轮数上能以明显高于均匀分布下的概率导致某输出差分值，这样的输入差分值和输出差分值就构成一个区分器，将约减轮的密码算法与随机函数区分开来。一条高概率差分迹或高概率差分闭包都可以称为差分区分器，通

常差分闭包中只有少数对差分概率起决定作用的差分迹。在区分器的前后各添加若干轮，攻击者可以选择并加密一定量的明文对使其尽量满足区分器差分，猜测在添加的轮数中对获得区分器输入差分值和输出差分值起作用的子密钥比特，正确的猜测将会使明显较多的明密文对经部分加解密后满足区分器，从而以少于穷搜的复杂度恢复密钥。这就构成了基本的差分分析。如果一个差分区分器的概率为 p，则得到一对能够满足该差分区分器的消息平均所需消息对的数量为 $\frac{1}{p}$。所以差分区分器的概率应尽可能大，使得攻击的数据复杂度和计算复杂度尽量减小。

为了抵抗差分攻击，对称密码算法需要使其最大差分概率尽可能接近随机函数的差分概率。最大差分概率指标也被用于评估密码算法抗差分攻击安全性的能力。线性操作只存在概率为 1 的差分和概率为 0 的差分，对于计算密码算法的高概率差分起决定作用的是非线性操作，如密码算法中普遍使用的 S 盒操作。对于 S 盒操作，为方便描述，我们给出下面的差分的输入/输出解集和可行输入/输出差分的定义。

定义 2.1 (差分的输入/输出解集). 给定 S 盒 $\mathrm{S}(\cdot)$，输入差分 δ_{in}，输出差分 δ_{out}，称集合 $V = \{x : \mathrm{S}(x) \oplus \mathrm{S}(x \oplus \delta_{in}) = \delta_{out}\}$ 为差分 $\delta_{in}, \delta_{out}$ 在 S 盒 $\mathrm{S}(\cdot)$ 下的输入解集。相应地，称集合 $\mathrm{S}(V) = \{\mathrm{S}(x) : \mathrm{S}(x) \oplus \mathrm{S}(x \oplus \delta_{in}) = \delta_{out}\}$ 为输出解集。

显然，差分的输入解集与输出解集在 S 盒下是一一对应的。在无歧义的情况下，可不加区别地称为差分解集。

定义 2.2 (可行输入 (输出) 差分). 给定 S 盒 $\mathrm{S}(\cdot)$ 和输入差分 δ_{in} (输出差分 δ_{out})，使得差分解集非空的输出 (输入) 差分称为可行输出 (输入) 差分。

对于非线性操作部件，其差分性质可以用差分分布表 (Differential Distribution Table, DDT) 来展现，表中的元素为相应差分解集中元素的个数。差分分布表中只有可行输入/输出差分下才是非零元素。差分分布表可以用穷搜的方法获得。表 2.1 给出了按位与 ($\wedge$) 操作作为二进一出的 S 盒的差分分布表。

S 盒的设计应尽可能满足差分均匀性，但是因为固有的性质，对诸如 4-比特、8-比特的 S 盒来说完全的均匀性是不可能达到的。4-比特 S 盒所能达到的最佳差分均匀性是使得最高差分概率为 2^{-4}，8-比特 S 盒为 2^{-6}。

分组密码由于每轮有轮密钥的参与，每轮的输入值通常视为独立的，那么整体差分概率即可通过每个活跃非线性部件上的差分概率的乘积计算。对于非

表 2.1: 按位与差分分布表

Δ_{in} \ Δ_{out}	0	1
(0,0)	4	0
(0,1)	2	2
(1,0)	2	2
(1,1)	2	2

线性部件仅为 S 盒的密码算法而言，高概率差分意味着差分中涉及了更少的活跃 S 盒。寻找高概率差分通常通过搜索非平凡差分中所涉及的最小活跃 S 盒个数进行。若最小活跃 S 盒个数的下界为 t，S 盒的最大差分概率为 2^{-w}，则密码算法的最高差分概率不会高于 $(2^{-w})^t$。t 的值越高，得到的最小活跃 S 盒个数下界就越紧，对密码算法安全性的评估也越准确。

差分分支数是描述差分在线性操作上扩散的指标。设 $L:\mathbb{F}_{2^\omega}^m \to \mathbb{F}_{2^\omega}^l$ 是一个从 m 个 $\mathbb{F}_{2^\omega}$ 分支到 l 个 $\mathbb{F}_{2^\omega}$ 分支的线性映射，$w_{\mathbb{F}_{2^\omega}}(x)$ 表示 x 中非 0 的 $\mathbb{F}_{2^\omega}$ 分支的数量，则 $L(\cdot)$ 的差分分支数定义为

差分分支数

$$\mathcal{B} = \min_{\Delta x \neq 0} \{w_{\mathbb{F}_{2^\omega}}(\Delta x) + w_{\mathbb{F}_{2^\omega}}(L(\Delta x)) : \Delta x \in \mathbb{F}_{2^\omega}^m\}.$$

差分分支数越大表示差分的扩散性越好。异或操作作为两个输入分支、一个输出分支的线性变换，其差分分支数为 2。$m \times m$ 维线性变换矩阵所能达到的最大差分分支数为 $(m+1)$，满足这一差分分支数的矩阵也称为 MDS (Maximum Distance Separable，极大距离可分) 矩阵，被广泛应用于密码设计。

2.3.2　高概率差分的搜索

搜索高概率差分迹或由多条差分迹组成的差分闭包是进行差分分析的第一步。近年来出现的自动化搜索方法因为实现简单、适用面广、可借助第三方软件求解等优点，成为广泛使用的方法。Mouha 等的基于混合整数线性规划的自动化搜索方法 [93] 是较早用自动化的方法求解最小活跃 S 盒个数的方法，后来得到了发展并用于高概率差分迹的搜索 [52, 105, 117, 119]。

线性规划问题是运筹学优化领域的一个经典问题，一个优化问题算例由一个需要最大化或最小化的目标函数和对函数中变量的一组约束条件构成，求解

优化问题即在这组约束下求解目标函数的最优值。若目标函数和约束条件都是线性的，就称为线性规划问题。根据涉及变量的类型，又有整数线性规划、0-1 规划、混合整数线性规划等分类。目前有很多软件和程序语言包用于求解优化问题，如 Gurobi [60]，SCIP [53] 等。

下面我们给出用线性规划求字级密码最小活跃 S 盒个数的基本方法 [93]。对字级密码，通常 S 盒的输入即为一个字级分支。首先定义差分变量：

差分变量 将加密过程中每个位置处的字级差分用一个 0-1 变量 x_i 表示，称为这个位置的差分变量。x_i 的取值满足

$$x_i = \begin{cases} 0, & \text{若该位置差分为 } 0, \\ 1, & \text{若该位置差分非 } 0. \end{cases} \tag{2.1}$$

S 盒的输入、输出差分变量取值是相同的，可用同一个变量表示。

然后设定目标函数，并将差分在各个操作部件上的传播规则用线性约束描述出来。

目标函数 所有 S 盒输入变量之和。

对异或操作的描述 设 α, β 为两个输入差分变量，γ 为输出差分变量。只有两个输入差分变量都取 0 时，输出差分变量才取 0，否则输出差分变量既可取 1 (两输入差分值不同) 又可取 0 (两输入差分值相同)。为描述这样的差分传播，需要引入辅助变量 $d_\oplus$，则线性约束条件为

$$\begin{aligned} \alpha + \beta + \gamma - 2d_\oplus &= 0, \\ d_\ominus - \alpha &\geq 0, \\ d_\ominus - \beta &\geq 0, \\ d_\oplus - \gamma &\geq 0. \end{aligned} \tag{2.2}$$

对线性变换的描述 设差分分支数为 $\mathcal{B}$ 的 $m \times m$ 维线性变换的输入差分变量为 $(x_0, x_1, \cdots, x_{m-1})$，输出差分变量为 $(y_0, y_1, \cdots, y_{m-1})$，引入辅助变量 d_L，

则线性变换上的差分传播约束条件为

$$\begin{aligned} x_0 + x_1 + \cdots + x_{m-1} + y_0 + y_1 + \cdots + y_{m-1} &\geq \mathcal{B} \cdot d_L, \\ d_L &\geq x_0, \\ d_L &\geq x_1, \\ &\cdots\cdots \\ d_L &\geq x_{m-1}, \\ d_L &\geq y_0, \\ d_L &\geq y_1, \\ &\cdots\cdots \\ d_L &\geq y_{m-1}. \end{aligned} \tag{2.3}$$

其他约束条件 为了防止全 0 情况出现，至少应有一个 S 盒活跃。另外，所有变量都置为 0-1 变量。

用优化软件求解得到的线性规划问题，可以得到密码算法的最小活跃 S 盒个数，从变量取值中可以得到截断差分迹。对密码算法进行比特级的建模，可以得到差分迹的具体形式 [52, 105, 117, 119]。

2.4　小 S 盒的硬件实现技术

S 盒是密码算法中的重要部件，好的 S 盒不仅要满足安全性要求，也要能进行高效的软硬件实现。S 盒的每个输出比特都可以看做输入比特的布尔函数，因此 S 盒的实现也可以看作布尔函数的实现。绝大多数 CPU 架构中包含的操作指令有 AND、OR、XOR、NOT，用这些指令对 S 盒的实现可以直接对应软件实现。硬件实现下还可以包含 NAND、NOR、XNOR 指令。本文关注的是可以使用所有七个逻辑门的面向硬件的实现方式。所有的逻辑门都可以只用 NAND 门或只用 NOR 门实现，因此将 NAND 门或 NOR 门用作“门等价” (gate equivalence, GE) 单位，也用作逻辑门的实现单位。NOT 门的实现需要 0.5 GE，AND 门、OR 门的实现需要 1.5 GE，XOR 门、XNOR 门的实现需要 2 GE [12]。

电路深度和电路面积是硬件实现中的两个重要指标。电路面积是计算一个给定布尔函数所需的最少的门的数量。电路深度是从输入门到输出门的最长

线路长度，可以充分考虑门的并行。逻辑门的 GE 实现既可以用作电路深度单位，也可以用作电路面积单位。小的电路深度意味着少的算法延迟，也就意味着提高算法的时钟频率 (clock frequency)。所有的布尔函数都可以在 2 个逻辑门的深度之内实现出来，比如将函数转化为 CNF 形式，但是这样会使电路面积增大，并不是理想实现方式。硬件实现的优化实际是在电路深度和电路面积之间进行折中处理的过程。

第三章　SHA-3 杂凑函数的约减轮碰撞攻击

Keccak 杂凑函数 [20] 自 2008 年提交至 SHA-3 竞赛以来受到了密码学者的极大关注 [10, 41, 46, 48, 50, 67, 87, 95]。2012 年，美国国家标准与技术研究所（National Institute of Standards and Technology, NIST）宣布 Keccak 为 SHA-3 竞赛的优胜算法并出台 FIPS 标准 [100] 作为新的 SHA-3 标准。SHA-3 标准包含四个输出长度固定的杂凑函数和两个可扩展输出长度的杂凑函数（eXtendable-Output Functions, XOFs）。四个输出长度固定为 d 的杂凑函数记为 Keccak-d，其中 $d = 224, 256, 384, 512$。两个可扩展输出长度的杂凑函数为 SHAKE128 和 SHAKE256，其输出摘要的长度可以扩展为任意需要的长度，后缀 128 和 256 仅表示其一般情况下能达到的安全强度。SHA-3 标准中的杂凑函数其内部状态的大小都为 1600 比特。在 Keccak 碰撞攻击和原像攻击挑战赛中 [18]，输出摘要的长度又有 160 和 80 的版本，且每个输出长度版本又对应四个内部状态大小分别为 200, 400, 800, 1600 的版本。

本章主要关注对 Keccak 系列的碰撞攻击。前人对 Keccak 系列最好的碰撞攻击结果是 2012 年由 Dinur 等人发表的对 Keccak-224 和 Keccak-256 的 4 轮碰撞攻击 [45] (后提供期刊版 [47])。在此之后，对 5 轮的 Keccak-256 碰撞攻击有了理论结果 [46]，但是能够实现实际碰撞攻击的结果仍然局限在 4 轮。为了促进对 Keccak 系列的安全性分析，Keccak 设计团队在 Keccak 挑战赛中提供了更多小版本算法以供分析，包括面向碰撞攻击的 160 比特输出长度的版本和面向原像攻击的 80 比特输出长度的版本，每个版本都包括四种内部状态大小，接受的攻击轮数也从 1 轮到 12 轮不等。这些版本抗碰撞攻击和原像攻击的理想安全性强度为 2^{80}，这比 SHA-3 标准中的四个版本安全强度更低，但是仍然超过了目前可有的计算资源能达到的计算能力。Dinur 等人 [45] 和 Mendel 等人 [87] 给出了部分挑战赛版本的 4 轮攻击。对 Keccak-256 的 5 轮理论分析结果由 Dinur 等 [46] 给出，其复杂度为 2^{115}，运用了扩展的内部差分。这是已知的在本文工作之前对 5 轮及 5 轮以上 Keccak 的仅有分析结果。

我们研究了代数与差分相结合的方法来进行 Keccak 的碰撞攻击，实际找到了 5 轮 SHAKE128 和两个 5 轮 Keccak 挑战赛版本的碰撞。我们也将 Dinur 等的对 Keccak-224 和 Keccak-256 的 4 轮碰撞攻击复杂度进一步降低。

这些结果得自于对 KECCAK S 盒的一个重要观察：当输入值限制在一些仿射子空间时，KECCAK 的 S 盒可以被表示为线性变换，且能够使 S 盒线性化的子空间的维数最大为 2。Dinur 等人已经应用了当 KECCAK S 盒的输入差分和输出差分固定时，差分解集构成了一个维数最多为 3 的仿射子空间。我们发现其中维数为 2 的仿射子空间都是可以使 S 盒线性化的仿射子空间，而对于其中每个维数为 3 的仿射子空间，都包含有六个 2 维仿射子空间使得 S 盒可以线性化。基于这个性质，我们将 KECCAK 的第一轮 S 盒全部线性化，这样第一轮整体便成为线性变换。再结合将第二轮 S 盒层逆向的方法，我们把建立一个 2 轮连接器的问题转化为求解一个线性方程组。一次性求解线性方程组可以产生大量的解，使得其中至少有一个可以满足后面 3 轮或更多轮的差分迹。

线性化 S 盒的一个负面作用是使得方程组的自由度迅速下降，从而使得 2-轮连接器不一定存在。为了克服这个难题，在找差分迹时，尽量找给 2- 轮连接器带来的约束最少的差分迹。实际的实验结果验证了方法的正确性，我们找到了 5 轮 SHAKE128 和两个 5 轮 KECCAK 挑战赛版本的实际碰撞。

本文结果和相关工作的对比在表 3.1。建立 2-轮连接器的算法是启发式的，其求解复杂度并没有一个理论估计，因此用实际的运行时间来代替。 本章内

表 3.1: KECCAK 碰撞攻击结果及比较

版本$[r,c,d]$	n_r	碰撞搜索复杂度	碰撞搜索时间	方程求解时间[1]	参考文献
SHAKE128	5	2^{39}	30m	25m	3.3.1
KECCAK[1440,160,160]	5	2^{40}	2.48h	9.6s	3.3.2
KECCAK[640,160,160]	5	2^{35}	2.67h	30m	3.3.3
KECCAK-224	4	2^{24}	2∼3m		[45]
		2^{12}	0.3s	2m15s	3.3.4
KECCAK-256	4	2^{24}	15∼30m		[45]
		2^{12}	0.28s	7m	3.3.4

[1] 方程求解复杂度没有理论估计。

容安排如下：3.1 节给出了对 KECCAK 算法的具体描述。3.2 节给出了 2- 轮连接器的代数构造和详细的攻击方法。3.3 给出了实际的攻击实验数据和结果。3.4 节对本章进行总结。

3.1 KECCAK 杂凑函数

在本节，我们给出 KECCAK 杂凑函数的简要描述。所用到的主要符号如下：

c	Sponge 函数的内部状态大小
r	Sponge 函数的外部状态大小
b	Sponge 函数的宽度, $b = r + c$
d	输出消息摘要长度
p	填充消息块被约束的比特数, KECCAK 中 $p = 1$, SHAKE 中 $p = 6$
n_r	轮数
$\theta, \rho, \pi, \chi, \iota$	构成一轮的五步操作, 下标 i 表示该操作在第 i 轮, 例如 θ_i 表示第 i 轮的 θ 层， $i = 0, 1, 2, \cdots$
L	θ, ρ, π 的复合
L^{-1}	L 的逆操作
RC	轮常数
$\mathrm{S}(\cdot)$	5-比特 S 盒操作
$\mathrm{R}^i(\cdot)$	前 i 轮的 KECCAK 置换
δ_{in}	S 盒的 5-比特输入差分
δ_{out}	S 盒的 5-比特输出差分
DDT	差分分布表
α_i	第 i 轮轮函数的输入差分, $i = 0, 1, 2, \cdots$
β_i	第 i 轮 χ 层的输入差分, $i = 0, 1, 2, \cdots$
w_i	第 i 轮差分概率的重量, $i = 0, 1, 2, \cdots$
$m_1 \| \| m_2$	字符串 m_1和 m_2 的连接
x	第一轮 χ 层之前的比特值
y	第一轮的输出比特值
z	第二轮 χ 层之前的比特值

KECCAK 杂凑函数应用了 Sponge 结构，对消息的处理分为“吸收”阶段和“挤出”阶段，如图 2.5 所示。消息首先用首尾为 1 中间为若干个 (可为 0 个) 0 的字符串 10^*1 进行填充，使其成为外部状态大小 r 的倍数。将初始消息记为 M，将填充后的消息记为 $\overline{M} = M||10^*1$。b 比特的状态初始化为全 0 状态。在吸收阶段，填充后的消息分成 r 比特的消息块，将消息块异或到目前内部状态的前 r 比特中，然后对整个 b 比特内部状态进行 f 置换操作，该置换也记为 KECCAK-f。依次处理每个消息块，当所有消息块都处理完毕后，进入到挤出阶段。在挤出阶段，内部状态的前 r 个比特每输出一次，进行一次 f 置换操作，直到所需要的 d 比特全部输出。状态大小 $b = 1600$，内部状态 $c = 2d$ 的 KECCAK 记为 KECCAK-$d, d = 224, 256, 384, 512$，是 SHA-3 标准中所采用的函数。此外 SHA-3 标准中还有两个函数 SHAKE128 和 SHAKE256，分别定义自 KECCAK-128 和 KECCAK-256，即其内部状态大小 c 分别为 256 和 512，但其输出摘要长度分别至少为 256 比特和 512 比特，且在对给定消息 M 进行填充前需要先在 M 后添加四比特全 1 后缀 1111。无特殊说明时，本文 SHAKE128 和 SHAKE256 的输出摘要长度即为 256 和 512。我们仍用 $\overline{M}$ 来表示在 SHAKE 填充过的消息。KECCAK 挑战赛中的版本将根据其参数显式地记为 KECCAK$[r, c, n_r, d]$。

在 SHA-3 标准中，置换 KECCAK-f 包含 24 轮，每轮由 5 个操作作用在 1600 比特的内部状态上，内部状态可以表示为 5×5 个比特串。用 l 来表示单个比特串长度，A 表示 $5 \times 5 \times l$ 比特的内部状态，则每个比特可以表示为 $A[i, j, k], 0 \le i < 5, 0 \le j < 5, 0 \le k < l$。单个维度上的比特串分别定义和表示为行 $A[*][j][k]$、列 $A[i][*][k]$ 和巷 $A[i][j][*]$。

每轮置换的五个操作为：

$\theta : A[i][j][k] \leftarrow A[i][j][k] + \sum_{j'=0}^{4} A[i-1][j'][k] + \sum_{j'=0}^{4} A[i+1][j'][k-1]$。

$\rho : A[i][j][k] \leftarrow A[i][j][k + T(i,j)]$，其中 $T(i,j)$ 是既定常数。

$\pi : A[i][j][k] \leftarrow A[i'][j'][k]$，其中 $\begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \cdot \begin{pmatrix} i' \\ j' \end{pmatrix}$。

$\chi : A[i][j][k] \leftarrow A[i][j][k] + ((\neg A[i+1][j][k]) \wedge A[i+2][j][k])$。

$\iota : A \leftarrow A + RC[i_r]$，其中 $RC[i_r]$ 是轮常数。

当 $l = 8, 16, 32$ 时，对应的内部状态大小分别为 $200, 400, 800$，相应置换函数中的 ρ 操作也是在模对应的 l 值下进行的。这些内部状态缩小的版本为 KECCAK 挑战赛版本。

置换 KECCAK-f 的轮函数中前三层操作是线性映射，它们的复合记为 $L \triangleq \pi \circ \rho \circ \theta$。轮函数中唯一的非线性层是 χ，可以看做作用在状态的 $5 \cdot l$ 个行上的 5-比特 S 盒。用 $\mathrm{S}(x)$ 来表示对 5-比特输入值 x 的 S 盒替换。该 S 盒的差分分布表记为 DDT，$\mathrm{DDT}(\delta_{in}, \delta_{out})$ 是集合 $\{x : \mathrm{S}(x) \oplus \mathrm{S}(x \oplus \delta_{in}) = \delta_{out}\}$ 的大小。前 i 轮的 KECCAK 置换记为 R^i（每轮置换只有加入的轮常数不同，在忽略轮常数差异的情况下可以将其视为相同的变换），$\mathrm{R}^i(\overline{M})$ 即为填充消息 $\overline{M}$ 经过 i 轮置换后的状态。

3.2 约减轮碰撞攻击

本章节首先给出碰撞攻击的方法概述，然后给出构造两轮连接器的具体代数方法。如无特殊说明，在本文攻击中使用的消息在填充后只有一个长度为 r 的消息块。为了符合 KECCAK 的填充规则，填充消息的最后一比特应置为 1，而前面的 $r-1$ 比特可由攻击者完全掌控，内部状态的 c 个比特根据 KECCAK 的初始化要求应置为全 0。当分析 SHAKE 时，有 $r-6$ 个比特可以由攻击者掌控，填充消息的最后 6 个比特根据 SHAKE 的消息处理和填充规则应置为全 1。

沿用 Dinur 等 [45] 及其他利用差分迹进行碰撞攻击的方式，我们的碰撞攻击由两部分构成：高概率差分迹和一个可以连接差分迹和初始值的连接器，如图 3.1 所示。令 ΔS_I 和 ΔS_O 分别表示差分迹的输入差分和输出差分。Dinur 等开发了他们称为“目标差分算法”的方法来找到消息对 (M, M') 满足经过一轮置换后的差分恰好为 ΔS_I，即 $\mathrm{R}^1(\overline{M}||0^c) + \mathrm{R}^1(\overline{M'}||0^c) = \Delta S_I$。下面，我们用一个代数方法将这个连接器扩展到两轮，即研制一个新的目标差分算法来找到消息对 (M, M') 满足经过两轮置换后的差分恰好为 ΔS_I，即 $\mathrm{R}^2(\overline{M}||0^c) + \mathrm{R}^2(\overline{M'}||0^c) = \Delta S_I$。大量这样的消息对以一定的概率满足从 ΔS_I 到 ΔS_O 的差分迹，如果 ΔS_O 的前 d 比特为 0，则能找到碰撞。因为攻击要以低复杂度进行，所以连接器求解的过程应当具有常数的或实际的复杂度，这样这部分的复杂度在整个碰撞攻击的复杂度中将不会起决定作用。获取 $\Delta S_I \rightarrow \Delta S_O$ 差分的方法将在第 3.2.4 节中给出。

从整体上看，两轮连接器的约束条件有：

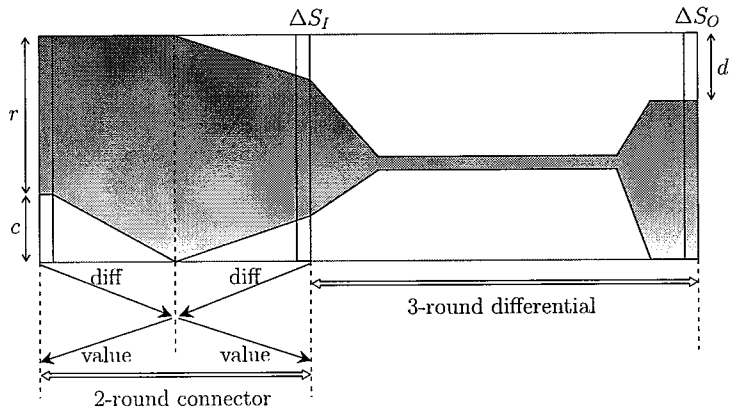


图 3.1: 5-轮 KECCAK 碰撞攻击示意图

1. 初始状态的最后 $c+1$ （或 $c+6$）比特固定；

2. 两轮置换之后的输出差分给定且固定 (该差分由使用的差分迹决定)。

以初始状态的前 $(r-1)$ (或 $(r-6)$) 比特的自由度为起始，将自由度作为一个衡量求解的有效性的量。下面将从对 KECCAK S 盒的观察出发，给出两轮连接器的构造和求解算法。

3.2.1 S 盒的线性化和仿射子空间

KECCAK 的状态大小比摘要长度大得多，这就给攻击者提供了极大的自由度。攻击者可以选择有效空间中某个满足特殊性质的子集来进行快速求解。在 KECCAK 中，我们选择可以使 S 盒线性化的子集，即当输入限制在这个子集上时，S 盒可以用线性变换来表示。显然，当考虑整个 $\{0,1\}^5$ 输入空间的时候 S 盒是非线性的，代数次数大于 1。但是，如下文所示，可以找到 2 维的仿射子空间，使得在该仿射子空间上 S 盒是可以线性化的。注意到 S 盒是 KECCAK 轮函数中唯一的非线性部件，因此当相应位置的值限制在这样的仿射子空间上时，整个轮函数等价于线性变换。

定义 3.1 (可线性化仿射子空间). 使得定义在该仿射子空间上的 S 盒变换等价于一个线性变换的仿射子空间叫做可线性化仿射子空间。若 V 是 S 盒 $\mathrm{S}(\cdot)$ 的可线性化仿射子空间，则 $\forall x \in V, \mathrm{S}(x) = A \cdot x \oplus b$，其中 A 是一个矩阵，b 是常数向量。

例如，当输入限制在子集 {00, 01, 04, 05} 上时，相应的 KECCAK S 盒的输出为 {00, 09, 05, 0C}，则 S 盒可以重新表示成线性变换：

$$y = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot x \tag{3.1}$$

其中 x 和 y 是输入值和输出值的向量表示，最右侧比特在顶端。

对 KECCAK 的 S 盒的可线性化仿射子空间有如下结论成立：

命题 3.1. *在 S 盒的 5-维输入空间 $\{0,1\}^5$ 中，*

1. *有 80 个 2-维可线性化仿射子空间，如附件中表 A.1 所列。*
2. *不存在 3-维及 3-维以上的可线性化仿射子空间。*

任意 1-维的子空间显然也是可线性化的仿射子空间。

由于仿射子空间要和差分迹一起使用，我们考虑 S 盒的可行输入、输出差分解集对应的可线性化仿射子空间，这和 S 盒的差分分布表 (DDT) 有着密切联系。KECCAK S 盒的差分分布表在附件 A.2 中。我们有如下结论：

命题 3.2. *给定一个 5-比特输入差分 δ_{in} 和一个 5-比特输出差分 δ_{out}，满足 $\mathrm{DDT}(\delta_{in}, \delta_{out}) \neq 0$，记满足该差分的输入值的集合为 $V = \{x : \mathrm{S}(x) + \mathrm{S}(x + \delta_{in}) = \delta_{out}\}$，输出集合为 $\mathrm{S}(V) = \{\mathrm{S}(x) : x \in V\}$。我们有*

1. *如果 $\mathrm{DDT}(\delta_{in}, \delta_{out}) = 2$ 或 4，那么 V 是一个可线性化仿射子空间。*
2. *如果 $\mathrm{DDT}(\delta_{in}, \delta_{out}) = 8$，那么存在六个 2-维子集 $W_i \subset V, i = 0, 1, \cdots, 5$，使得 $W_i (i = 0, 1, \cdots, 5)$ 是可线性化仿射子空间。*

值得注意的是从 DDT 中分析得到的 2-维可线性化仿射子空间涵盖了引理 3.1 中所述的所有 80 个 2-维仿射子空间。在文献 [57] 中已经注意到仿射子空间与 4- 比特 S 盒的差分分布表中值为 2 和 4 的元素之间的对应关系。对于 KECCAK S 盒的差分分布表中值为 8 的元素，后面将随机使用这六个可选的 2-维

可线性化仿射子空间。例如，对应 DDT(01, 01) 的 3-维仿射子空间，即满足输入差分为 1 且输出差分也为 1 的子空间是 {10, 11, 14, 15, 18, 19, 1C, 1D}，则包含于它的 6 个 2- 维可线性化仿射子空间为：

$$\begin{aligned}&\{10,11,14,15\},\\&\{10,11,18,19\},\\&\{10,11,1C,1D\},\\&\{14,15,18,19\},\\&\{14,15,1C,1D\},\\&\{18,19,1C,1D\}.\end{aligned}\tag{3.2}$$

考虑 KECCAK 的整个状态，对应每个 S 盒的仿射子空间的直积构成了整个状态的维数更大的仿射子空间。换句话说，当轮函数中所有的 S 盒都被线性化后，整个轮函数也就成为线性函数。这也是下面处理 2-轮连接器中第一轮的 S 盒层的方式。

3.2.2 2-轮连接器的构造及碰撞攻击算法

构造 2-轮连接器的核心是将问题转化为求解一个线性方程组。两轮的 KECCAK 置换可以表示为 $\chi_1 \circ L_1 \circ \chi_0 \circ L_0$ (忽略 ι)。给定 χ_0 层的输入差分和输出差分，用将输入值限制在可线性化仿射子空间的方法将 χ_0 层线性化，则 $L_1 \circ \chi_0 \circ L_0$ 变为线性变换。我们先给出如何用线性方程约束来将 χ_1 逆回，然后给出 χ_0 层的输入差分 β_0 的选择方法，进而给出连接器构造的完整算法。

χ_1 层的逆回即对 χ_1 的输入做限制使得 χ_1 层的差分以概率 1 满足。χ_1 的输出差分是 ΔS_I，即 3-轮差分迹的输入差分。χ_1 层的 S 盒并不是全都活跃的，我们只考虑活跃 S 盒，将每个活跃 S 盒随机地选择一个可行输入差分进行逆回。给定输出差分 δ_{out}，可行输入差分为 $\{\delta_{in} : \mathrm{DDT}(\delta_{in}, \delta_{out}) \neq 0\}$。如 Dinur 等 [45] 注意到的，对任意给定差分对 $(\delta_{in}, \delta_{out})$，解集 $V = \{x : \mathrm{S}(x) + \mathrm{S}(x + \delta_{in}) = \delta_{out}\}$ 构成一个仿射子空间。当解集 V 的大小为 2^{5-i} 时，可以通过建立 i 个二元域上的线性方程来从 $\{0,1\}^5$ 空间得到需要的解集 V。例如，对应于 DDT(03, 02) 的差分解集是 2-维仿射子空间 {14, 17, 1C, 1F}，它可以由下面的三个线性方程

来给出：

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot x = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}. \tag{3.3}$$

值得注意的是，在 i 个线性约束或集合 V 下，δ_{in} 和 δ_{out} 之间有一一对应的关系，即给定其中一个，另一个可以确定性地得出。因此，χ_1 的每个活跃 S 盒都可以选择一个可行的输入差分并用对应的 i 个线性方程约束输入值，从而以概率 1 逆回 χ_1 层。参照图 3.2，第 i 轮的输出差分为 α_i，初始状态差分为 α_0，$\beta_i = L(\alpha_i)$ 是 χ 层的输入差分。选择 χ_1 的可行输入差分也即选择 β_1，则 α_1 可以通过 $\alpha_1 = L^{-1}(\beta_1)$ 来唯一确定。下面将给出具体的实现方法。

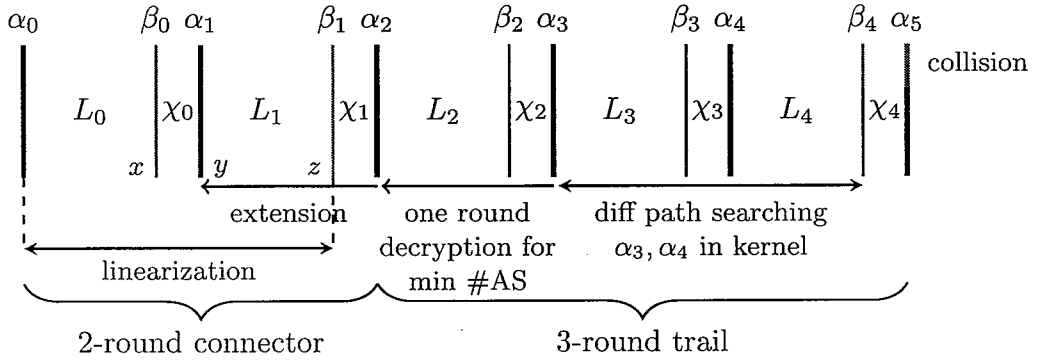


图 3.2: 5-轮 KECCAK 碰撞细节图.

如图 3.2 所示，方程组的自变量是 χ_0 层之前的比特值变量，用向量 x 表示。y 和 z 是中间变量，向量 y 表示 χ_0 的输出值变量，向量 z 表示 χ_1 之前的输入值变量。建立 2-轮连接器即构造关于 x 的线性方程组，需要把所有关于差分和仿射子空间的约束转移到变量 x 上。首先需要固定差分值 β_1 和 β_0。β_1 的固定即采用 χ_1 层逆回的方法，对 χ_1 层的每个活跃 S 盒随机选取可行的输入差分。对 β_0 的选取，我们基于 Dinur 等 [45] 的“目标差分算法”以更一般的方式来选取 β_0，方法分为两个阶段——差分阶段和数值阶段。β_1 选定后，建立两个方程组 E_Δ 和 E_M。E_Δ 用于对消息的差分的约束，E_M 用于对其中一个消息的值的约束。E_Δ 的初始化需要遵守：

(1) 消息填充约束，初始状态差分的最后 $c+p$ 个比特为 0；

(2) χ_0 的非活跃 S 盒的输入差分为 0。

E_M 的初始化需要遵守填充规则，即

- 初始状态的最后 $(c+p)$ 个比特值为 $1^p||0^c$。

其中，对 KECCAK-d 的攻击中 $p=1$，对 SHAKE 的攻击中 $p=6$。因为方程组变量 x 与初始状态之间呈线性关系，所以对初始状态的线性约束可以直接转化为对变量 x 的线性约束。因此，在初始化阶段，只需要在 E_Δ 和 E_M 中把相应的比特表达式赋予应有的值即可。

对 E_Δ 需要添加额外的方程来保证 α_1 是 β_0 的可行输出差分。最直观的方式是将 χ_0 层的每个活跃 S 盒的 5 比特输入差分值确定，但是这样会使得解空间迅速减小。参照文献 [45] 中的方法，对每个活跃 S 盒，首先选择可行输入差分集合中的 2-维仿射子空间来代替一个确定的输入差分值。这是基于对 KECCAK S 盒的一个观察：给定任意非零输出差分，所有可行输入差分构成的集合包含至少 5 个 (至多 17 个) 2-维仿射子空间。对每一个活跃 S 盒添加 3 个方程来将输入差分限制在一个 2-维仿射子空间中，这样求解 E_Δ 得到的解空间是 β_0 的候选仿射子空间。下一步，对每个活跃 S 盒，迭代地向 E_Δ 中添加使 5-比特输入差分唯一确定的另外 2 个方程，直到全部活跃 S 盒添加完毕且 E_Δ 保持一致性，此时 E_Δ 的解唯一，即为选定的 β_0 的值。通过这种方式，我们总能从 α_1 找到可行的 β_0，且满足初始化条件。由于此时活跃 S 盒的输入差分和输出差分已经唯一确定，可向 E_M 中添加将输入比特限制在相应差分解集中的方程。此时的 E_M 将作为下一步进行线性化的基础。

β_1 和 β_0 的值已唯一确定，ΔS_I (也即 α_2) 的值已给定，现在需要建立连通 2 轮的方程组。对于 χ_1 中的所有活跃 S 盒，根据 β_1 和 α_2 的值将输入值限制在相应的差分解集空间中，使得从 β_1 能以概率 1 得到 α_2，该限制记为

$$G \cdot z' = m, \tag{3.4}$$

可知 G 是一个分块对角矩阵，每个对角分块和 m 中的常数向量一起构成了对一个活跃 S 盒输入值的约束。z' 即为 z 中活跃 S 盒对应的变量。z' 和 y 之间有关系

$$\overline{L} \cdot (y + \overline{RC}[0]) = z', \tag{3.5}$$

其中 $\overline{L}$ 是 L 矩阵中对应 z' 的行，$\overline{RC}[0]$ 是第一轮的部分对应轮常数。将 $\overline{L}$ 中的全 0 列抽掉，记为 $\hat{L}$；相应地将 y 和 $\overline{RC}[0]$ 的对应行抽掉，记为 y' 和 $\widehat{RC}[0]$，我们可以得到关于 z' 的更紧凑的表达式

$$\hat{L} \cdot (y' + \widehat{RC}[0]) = z'. \tag{3.6}$$

只有 y' 所在的 S 盒才是第一轮需要线性化的 S 盒。将第一轮中需要线性化的 S 盒的输入限制在可线性化仿射子空间中，则第一轮的 χ_0 层可以用线性映射替代。限制于可线性化仿射子空间的约束为

$$A \cdot x = t, \tag{3.7}$$

则 x 和 y' 可以通过线性映射联系起来：

$$\chi_L \cdot x + \chi_C = y',$$

其中 χ_C 是仿射子空间的常数偏移。注意方程 (3.7) 覆盖了 E_M 中将 x 约束于差分解集空间的方程。

综上方程，加到 z 上的约束可以用下面的方程转移到方程组的未知变量 x 上：

$$G \cdot \hat{L} \cdot (\chi_L \cdot x + \chi_C + \widehat{RC}[0]) = m. \tag{3.8}$$

再考虑到 x 需要满足的初始化条件，即初始状态的最后 $(c+1)$（或 $(c+6)$）比特固定，有已经添加到 E_M 中的方程

$$\overline{L^{-1}}(x) = CA, \tag{3.9}$$

其中 CA 即是关于内部状态比特和填充比特的预设值，$\overline{L^{-1}}$ 是 L^{-1} 矩阵的对应行。综上，E_M 包含 (3.7), (3.8) 和 (3.9)，E_M 的解空间即为满足 2-轮连接器的解空间。下面我们给出所述方法的算法实现步骤。

我们用线性化基本程序来产生用于将 x 限制在可线性化仿射子空间里的方程，使得第一轮的 χ_0 层能够被线性化；用线性化主体程序来产生用于使第二轮中 χ_1 层以概率 1 通过的方程。线性化基本程序的输入是 β_0，与确定的 β_0 相

对应的 E_M，第一轮的输出差分 α_1，和涉及到的需要线性化得到的第一轮的输出比特 y'。

线性化基本程序：

输入： $E_M, \beta_0, \alpha_1, y'$。

输出： 更新的 E_M, χ_L, χ_C。

1. 初始化一个矩阵 χ_L 和一个向量 χ_C。

2. 对 y' 的每个比特进行迭代，计算该比特在 S 盒尺度上的索引，不妨设为第一轮第 i 个 S 盒的第 j 个比特。则对对第一轮的第 i 个 S 盒：

   (a) 如果该 S 盒在本过程中没有被处理过：

   (i) 如果该 S 盒是非活跃的，则随机选取一个可线性化的 2-维仿射子空间，检测刻画该 2-维仿射子空间的 3 个方程是否与当前的 E_M 相一致。如果一致，将其添加到 E_M 中，并且更新 χ_L 和 χ_C：将该 2-维子空间上 S 盒的等价线性变换方程的第 j 行添加进去。继续处理第 2 步中 y' 的下一个比特。若不一致，选择另外一个可线性化的 2-维仿射子空间，检测一致性并根据是否一致更新 E_M、χ_L 和 χ_C。如果所有的可线性化 2-维仿射子空间都无法使方程组一致，输出“线性化基本程序无解”，结束。

   (ii) 否则，若该 S 盒活跃，则从 β_0 和 α_1 中找出其对应的输入差分和输出差分，分别记为 δ_{in} 和 δ_{out}。

   情况 1： $\mathrm{DDT}(\delta_{in}, \delta_{out}) = 8$。从六个可线性化 2-维仿射子空间中随机选取一个，获得刻画该仿射子空间的方程（三个方程中有两个方程已经含在 E_M 中了），如果当前 E_M 与该方程相一致，则将该方程添加进 E_M，并将 S 盒在该 2-维子空间上的等价线性映射的第 j 行添加进 χ_L 和 χ_C。继续处理第 2 步中 y' 的下一个比特。否则，若方程与当前 E_M 不一致，则继续选取另外的 2-维仿射子空间。如果全部六个 2-维可线性化仿射子空间对应的方程都无法与 E_M 相一致，则输出“线性化基本程序无解”，结束。

情况 2： $\mathrm{DDT}(\delta_{in}, \delta_{out}) = 2$ 或 4。将 S 盒在该差分解集空间上的等价线性映射的第 j 行添加进 χ_L 和 χ_C。继续处理第 2 步中 y' 的下一个比特。

(b) 否则，如果第 i 个 S 盒已经被处理过，则将已经选定的可线性化仿射子空间上的等价线性映射的第 j 行添加进 χ_L 和 χ_C。继续处理第 2 步中 y' 的下一个比特。

3. 输出当前的方程组 E_M 以及 χ_L 和 χ_C。此时的 χ_L 和 χ_C 满足 $\chi_L \cdot x + \chi_C = y'$。

线性化主体程序的输入是 $\beta_0, \alpha_1, \beta_1, \alpha_2(\Delta S_I)$ 和与确定的 β_0 相对应的 E_M。

线性化主体程序

输入： $E_M, \beta_0, \alpha_1, \beta_1, \alpha_2$。

输出： 更新的 E_M。

1. 利用 β_1 和 α_2，初始化方程 (3.4) 中的系数矩阵 G 和常数向量 m，使得方程 $G \cdot z' = m$ 将第二轮 S 盒层的输入值 z 限制在 β_1 和 α_2 对应的差分解集空间中。

2. 获得方程 (3.6) 的系数矩阵 $\hat{L}$ 和常数 $\widehat{RC}[0]$，以及 y'。满足 $\hat{L} \cdot (y' + \widehat{RC}[0]) = z'$。

3. 初始化计数器，置为 0。

4. 以 y' 和 E_M, β_0, α_1 为输入执行线性化基本程序。如果成功，则会获得第一轮 S 盒层线性化的结果，即有 $\chi_L \cdot x + \chi_C = y'$，然后进入第 6 步。否则，进入第 5 步。

5. 计数器值加 1。如果计数器值等于预设的阈值 $T1$，输出“失败”，结束程序。否则，E_M 置为输入值，进行第 4 步。

6. 计算方程组 (3.8)，检测方程组 (3.8) 是否与 E_M 相一致。如果一致，添加方程 (3.8) 到 E_M 中，输出最终 E_M。否则，进行第 5 步。

注意到线性化主体程序并非总是成功的，为了克服这个问题，我们从 3-轮差分迹的输入差分 ΔS_I 出发，随机选取不同的可行输入差分 β_1 直到程序成功。由于 α_2 中的活跃 S 盒个数足够大（几十到几百），有足够的 β_1 的不同选取方式使得程序成功的概率足够高。值得注意的一点是从 α_2 逆回到 β_1 的过程无需保持高概率，因为这个过程在构造的 2-轮连接器中被覆盖了，可以以概率 1 满足。另外，输入差分活跃 S 盒个数无需设限也为 3 轮差分迹的搜索提供了更多的自由度。

最后，从 E_M 的解空间里，搜索 3 轮的差分碰撞，即可得到 5 轮碰撞结果。

3.2.3 自由度分析

E_M 解空间的自由度是碰撞攻击能否成功的关键因素。自由度大于 3-轮差分迹重量的解空间才更可能包含碰撞消息对。第一轮线性化后，自由度为 $\sum_{i=0}^{\frac{b}{5}-1} \mathrm{DF}_i^{(1)}$，其中 $\mathrm{DF}_i^{(1)}$ 表示第一轮第 i 个 S 盒 5-比特输入值的自由度。其取值根据表 3.2 确定。

表 3.2: 自由度 $\mathrm{DF}_i^{(1)}$ 取值规则

$\mathrm{DF}_i^{(1)*}$	非活跃	$\mathrm{DDT}(\delta_{in},\delta_{out})$=2	$\mathrm{DDT}(\delta_{in},\delta_{out})$=4	$\mathrm{DDT}(\delta_{in},\delta_{out})$=8
y' 涉及	2	1	2	2
y' 未涉及	5	1	2	3

* $\mathrm{DF}_i^{(1)}$ 的取值决定于第 i 个 S 盒是否在 y' 中涉及以及输入差分 δ_{in} 和输出差分 δ_{out} 所对应的 $\mathrm{DDT}(\delta_{in},\delta_{out})$ 的值。

对初始状态的差分使自由度减少了 $(c+p)$。另外一部分自由度的减少是在对第二轮 S 盒输入值的约束上。第二轮 S 盒 5-比特输入值的自由度 $\mathrm{DF}_i^{(2)}$ 的取值为

$$\mathrm{DF}_i^{(2)} = \begin{cases} 1, & \mathrm{DDT}(\delta_{in},\delta_{out}) = 2, \\ 2, & \mathrm{DDT}(\delta_{in},\delta_{out}) = 4, \\ 3, & \mathrm{DDT}(\delta_{in},\delta_{out}) = 8, \\ 5, & \mathrm{DDT}(\delta_{in},\delta_{out}) = 0, \end{cases} \tag{3.10}$$

其中 δ_{in} 和 δ_{out} 是第二轮第 i 个 S 盒的输入差分和输出差分。对第二轮的每个 S 盒，向 E_M 中添加了 $(5-\mathrm{DF}_i^{(2)})$ 个方程，也认为自由度减少了相应的值。对

最终 E_M 解空间自由度的估计值为

$$\mathrm{DF} = \sum_{i=0}^{\frac{b}{5}-1} \mathrm{DF}_i^{(1)} - (c+p) - \sum_{i=0}^{\frac{b}{5}-1} (5 - \mathrm{DF}_i^{(2)}). \tag{3.11}$$

更大的 DF 值能够更加有利于两轮之后碰撞的搜索。

3.2.4　3-轮差分迹的搜索

本节主要介绍差分迹的搜索方法，主要遵循了文献 [41, 50, 95] 等的基于 Keccak 轮操作性质的核搜索方法。

θ, ρ, π, ι 是 Keccak 轮置换的线性操作，χ 是非线性操作。ι 对差分没有影响，ρ 和 π 不改变差分迹中活跃比特的个数。对差分迹起决定作用的是 θ 和 χ 操作。沿用 [20] 中的定义，将 $P(A)$ 定义为状态 A 的列奇偶性，即 $P(A)[i][k] = \Sigma_j A[i][j][k]$，值为奇数则称该列为奇的，值为偶数则称该列为偶的。若 A 的每一列都是偶的，则称 A 在 CP-核中。当 A 在 CP-核中时，θ 操作并不改变 A 的重量。一条 n 轮的差分迹可以表示为

$$\alpha_0 \xrightarrow{L} \beta_0 \xrightarrow{\chi} \alpha_1 \xrightarrow{L} \cdots \alpha_{n-1} \xrightarrow{L} \beta_{n-1} \xrightarrow{\chi} \alpha_n,$$

由于 α_i 和 β_i 之间的线性关系，也可以只用 α_i 或只用 β_i 来表示。χ_i 层的差分重量记为 w_i。对于 χ 操作有性质：给定一个 S 盒的输入差分，所有可行输出差分以相同概率出现；反之不然，但是可以确定概率最高的可行输入差分及其概率。因此，给定 β_i，$(\beta_i \to \alpha_{i+1})$ 的重量可以确定；反之，给定 β_i，$(\beta_{i-1} \to L^{-1}(\beta_i))$ 的最低重量可以确定。当输入差分重量为 1 时，输出差分与输入差分相同的概率为 2^{-2}。差分状态 α 中的活跃 S 盒个数用 $\#\mathrm{AS}(\alpha)$ 来表示。

参照文献 [20]，称 $(\beta_1, \cdots, \beta_{n-1})$ 为 n-轮的差分迹核，定义了一组 n-轮的差分迹 $\alpha_0 \xrightarrow{L} \beta_0 \xrightarrow{\chi} \alpha_1 \xrightarrow{L} \beta_1 \cdots \xrightarrow{L} \beta_{n-1} \xrightarrow{\chi} \alpha_n$，其中首轮由 $\alpha_1 = L^{-1}(\beta_1)$ 向前取最低重量，最后一轮 α_n 为可行差分。下面首先寻找好的 $(n-1)$-轮差分迹核。

用于攻击的差分迹需要满足的条件首先是输出摘要位置的差分为 0，记为 $\alpha_{n_r}^d = 0$。其次，为了满足自由度的要求，我们对方程 (3.11) 的前两项设置一个

阈值

$$\mathrm{TDF} = \frac{b}{5} \times 2 - (c + p),$$

使得差分迹满足

$$\mathrm{TDF} > w_1 + \cdots + w_{n_r-1}^{(d)}, \tag{3.12}$$

其中 $w_{n_r-1}^{(d)}$ 是 w_{n_r-1} 中与输出摘要相关的部分，当内部状态与外部状态衔接位置的 S 盒活跃 (仍保证外部状态部分差分为 0) 时取值非 0。值得注意的是，差分路径的搜索是在按照 3.2 节的方法构造 2-轮连接器之前进行的，而对差分路径搜索的精确要求来自于 2-轮连接器的求解结果（主要为自由度），在这样一种相互牵制的关系下，我们将 (3.12) 式的条件作为一个启发式条件来开始差分路径的搜索。

第三个要满足的条件就是碰撞搜索复杂度应在实际可行的范围内。2-轮连接器求解完成后，碰撞搜索的复杂度为 $2^{w_2+\cdots+w_{n_r-1}^{(d)}}$，因此将 $w_2 + \cdots + w_{n_r-1}^{(d)}$ 限制在 48 以内。

综上，为进行 5-轮的碰撞攻击，配合 2-轮连接器的构造，差分迹的搜索需要满足的三个条件为

(1) $\alpha_5^{(d)} = 0$，即输出差分为 0；

(2) $\mathrm{TDF} > w_1 + \cdots + w_4^{(d)}$，即 2-轮连接器的解空间自由度需要足够大；

(3) $w_2 + w_3 + w_4^{(d)} \leq 48$，即碰撞搜索复杂度需要足够小。

如文献 [41, 50, 95] 中所探究过的，不可能构造 3-轮的全部在 CP-核中的差分迹，但是可以构造 2-轮的 CP-核中的差分迹，来保证活跃比特尽量少地向前后传播。差分迹的搜索首先从低重量的 β_3 开始，利用 Keccak 官方网站提供的 Keccak 工具 [21] 搜索重量低于 8 且保证 α_3 和 α_4 都在 CP-核中的 β_3。对于每个获得的 β_3，穷尽所有可行的 α_4，计算 $\beta_4 = L(\alpha_4)$，此时即可判断是否有 $\alpha_5^{(d)} = 0$。保留满足这一条件的 β_3，由 $\alpha_3 = L^{-1}(\beta_3)$ 向前穷尽所有可行的 β_2，计算 $\#AS(\alpha_2)$，当该值足够小时，如小于 110，检测条件 (2)、(3) 是否满足。

利用搜索到的差分迹，配合 2-轮连接器，我们获得了部分 Keccak 杂凑函数 5 轮碰撞攻击的实际结果和 6 轮的理论结果，在下一章节具体说明。

3.3　攻击结果

3.3.1　5 轮 SHAKE128 碰撞攻击结果

将附录中表 A.3 中的第一条差分迹核用于 5 轮 SHAKE128 的碰撞攻击，我们选择了使得 $\beta_1 \to \alpha_2$ 满足最高概率的 β_1。经 25 分钟求解出 2-轮连接器的解后，解空间自由度为 94，足够以概率 2^{-39} 搜索后面 3 轮碰撞。搜索碰撞所用时间约为半小时。碰撞结果见附录中的表 A.5。

3.3.2　Keccak$[1440, 160, 5, 160]$ 碰撞攻击结果

将附录中表 A.3 中的第二条差分迹核用于 Keccak$[1440, 160, 5, 160]$ 的碰撞攻击，经 9.6 秒求解出 2-轮连接器的解，解空间自由度为 162，足够以概率 2^{-40} 搜索后面的 3 轮碰撞。搜索碰撞所用时间为 2.48 小时。搜索到的碰撞见表 A.4，该结果解决了 Keccak 挑战赛 [18] 的一个案例。

3.3.3　Keccak$[640, 160, 5, 160]$ 碰撞攻击结果

将附录中表 A.3 中的第三条差分迹核用于 Keccak$[640, 160, 5, 160]$ 的碰撞攻击，依旧选取使得 $\beta_1 \to \alpha_2$ 满足最高概率的 β_1。经 30 分钟求解出 2-轮连接器，解空间自由度为 56，经 2 小时 40 分钟搜索出概率为 2^{-35} 的后 3 轮碰撞。碰撞结果见附录中的表 A.6，该结果解决了 Keccak 挑战赛 [18] 的一个案例。

3.3.4　改进的 4 轮碰撞攻击结果

对 Keccak-224 和 Keccak-256 的 4-轮碰撞攻击已经在文献 [45] 中给出，但是用本章提出的方法可以将复杂度进一步降低。从 [45] 中提供的差分迹开始建立 2-轮连接器。对 Keccak-224，建立和求解 2-轮连接器的时间是 2 分钟 15 秒，搜索碰撞的复杂度仅为 2^{12}，用时 0.325 秒。对 Keccak-256，建立和求解 2-轮连接器的时间是 7 分钟，搜索碰撞的复杂度是 2^{12}，用时 0.28 秒。搜索复杂度均优于文献 [45] 中的 2^{24}。此外，文献 [45] 中指出，即使用他们的目标差分算法找到数量大于 2^{30} 的消息对集合，仍然不能保证在这个集合中可以搜索到碰撞。原因认为是前两轮置换的混淆作用不够充分，以及集合内消息之间的相关性。而利用本章的 2-轮连接器算法并没有出现这一问题，只要求得的 2-轮连接器的解空间足够大，总能从该解空间中搜索到可用的碰撞，无需再重复运行 2-轮连接器求解算法。

3.4 小结

本章给出了 SHA-3 杂凑函数 KECCAK 的约减轮碰撞攻击方法。该方法得益于 KECCAK 置换轮函数的非线性部件 S 盒在某些特定的仿射子空间上等价于一个线性变换的性质，那么通过对第一轮 S 盒的线性化，可以将前人构造的一轮连接器向前扩展一轮，以线性方程组的方式构造出 2-轮连接器。通过实验，该想法得到验证，我们实际找到了 5 轮的 SHAKE128 和两个 KECCAK 挑战赛案例的碰撞。其中对 SHAKE128 的攻击结果是对 SHA-3 标准杂凑函数轮数最高的实际攻击结果。

本章内容对应文献 [106]，感谢合作者的贡献。

第四章　基于整数规划的自动化密码分析方法及其在 FOX 密码分析中的应用

差分分析 [25] 是对对称密码算法的重要分析手段。当输入差分以非随机的方式产生输出差分时，可以对目标密码算法建立一个区分器，进而进行密钥恢复攻击。S 盒是对称密码算法设计中重要的非线性混淆部件，在差分分析中，由含有相同输入输出的差分迹组成的差分闭包的概率对差分分析的有效性起着重要作用。比如某算法最高概率差分闭包的概率不超过 p，则利用该差分闭包进行攻击所需的明文对数目平均意义下为 $\frac{1}{p}$。如果这个数量已经超过了密码本的规模或大于密钥穷搜的规模，那么这个算法就可以抵抗基本的差分攻击。差分迹中的活跃 S 盒个数决定了差分迹概率。因此，在分组密码的抗差分安全性评估中，通常把最小活跃 S 盒个数作为一个重要指标。

2009 年，Borghoff 等将基于混合整数线性规划 (Mixed-Integer Linear Programming, MILP) 的技术用于密码分析 [32]。他们将描述流密码 Bivium 的二次方程转化为 MILP 问题来恢复内部状态。特别地，他们提供了将布尔方程转化成线性方程组的技术。本章也采用了相同的技术来改进分析方法。

2011 年，Mouha 等 [93] 提出了用基于 MILP 的技术来自动化地计算分组密码和流密码的活跃 S 盒个数下界的方法，并且用于流密码 Enocoro-128v2 和分组密码 AES。2013 年至 2014 年，Sun 等 [117, 119] 将这一方法由 Mouha 的字节级推广到比特级，给出了在 SIMON，PRESENT，DES/DESL，LBlock 等更多算法上的应用。

本章的目标算法是分组密码 FOX。FOX 分组密码算法采用了 Lai-Massey 结构，在不同平台上的实现有很大的灵活性，且有较高的实现效率和安全级别 [68]。FOX 有两个版本，FOX64 和 FOX128，分别对应不同的分组长度和密钥长度。前人对 FOX 分组密码的分析主要有积分分析 [135]，不可能差分分析 [94, 136]，差错分析 [83] 和全子密钥分析 [62, 63]。文献 [68] 给出了 FOX 分组密码抵抗差分分析的安全性估计，给出了单密钥模型下的差分迹概率的上界。

当在 FOX 密码算法的两个版本 FOX64 和 FOX128 上应用 Mouha 等的基于 MILP 的技术时，我们发现，不管轮数如何变化，得到的最小活跃 S 盒个数分

别总是 5 和 9。主要的原因就是 Mouha 等的方法中描述异或操作的约束并不能反映比特差分的传播，而只是限定了作为字级差分的分支数为 2。如果考虑截断差分内部比特差分的关系，有些作为截断差分的差分迹是无效的。在 Sun 等的工作中 [117]，对异或操作的比特级约束依然沿用作为字级差分的约束，这显然是不合理的。在本章，我们将改进 Mouha 等和 Sun 等的基于 MILP 的方法中关于异或操作的约束，进而提升基于 MILP 方法的有效性。

本章我们在基于 MILP 的方法中使用新的比特级约束来精确描述差分在异或操作上的传播。新的约束反映了输入差分值的异或运算，对文献 [93] 和 [117] 中只包含分支数信息的条件进行了改进，合理缩小了线性规划问题的可行域。利用改进后的模型，我们得到了 1-6 轮 FOX64 的更紧的差分活跃 S 盒下界，得到 6 轮 FOX64 的最大差分迹概率的上界是 2^{-64}，从而证明了 6 轮 FOX64 即可抵抗基本的单密钥差分攻击，而非此前认为的 8 轮。对 FOX128, 得到了 5 轮内的差分活跃 S 盒个数下界，与文献 [68] 中的结果相一致。

本章的安排如下：4.1 节介绍了 FOX 分组密码算法和目前已知的分析。4.2 节给出了 Mouha 等和 Sun 等的利用基于 MILP 的技术计算差分活跃 S 盒个数的方法，然后用于 FOX 密码，发现了一类无效的差分传播形式。在 4.3 节，我们改进了 MILP 模型中对异或操作的约束条件，并且给出了将改进后的方法应用于 FOX64 和 FOX128 得到的结果。4.4 节对本章进行了总结。

4.1 FOX 分组密码

FOX 分组密码算法 [68] 由 Junod 等在 2005 年提出。作为迭代型分组密码，其轮函数采用了 Lai-Massey 结构 (LM 结构) [78, 79]。LM 结构首先由来学嘉和 Massey 在 1991 年提出。1999 年，Vaudenay [123] 将正形函数引入到 LM 结构并且证明改进后的 3 轮 LM 迭代具有随机性，改进后的 4 轮 LM 结构具有超伪随机性。

FOX 分组密码包含多个版本，可记为 FOX64/k/r 和 FOX128/k/r，其中 64 和 128 分别代表分组长度，r 和 k 分别代表轮数和密钥长度，可根据使用和实现平台的不同灵活改变。本节给出 FOX64 和 FOX128 的概括性描述，对 FOX 系列算法的详细描述可参考文献 [68]。

4.1.1　FOX64 和 FOX128 密码算法的描述

FOX64 的密钥长度为 128 比特，总迭代轮数为 16 轮，轮函数如图 4.1 和图 4.2 所示。轮函数中的正形函数 or 将 32-比特的输入 $X_{(32)} = X_{0(16)}||X_{1(16)}$ 映射成 32-比特的输出 $Y_{(32)} = Y_{0(16)}||Y_{1(16)} = X_{1(16)}||(X_{0(16)} \oplus X_{1(16)})$。混淆函数 mu4 将输入视为 $\mathbb{F}_{2^8}$ 上的 4-维向量，将它与分支数为 5 的矩阵相乘得到输出。S 盒在最大差分概率为 $\mathrm{DP}_{\max}^{\mathrm{sbox}} = 2^{-4}$。

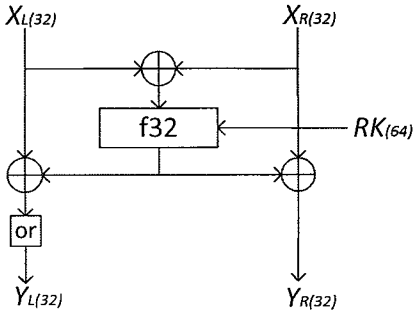


图 4.1: FOX64 轮函数

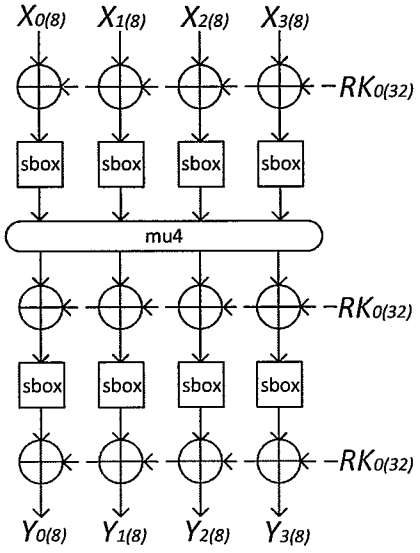


图 4.2: FOX64 中函数 f32

FOX128 的密钥长度为 256 比特，总迭代轮数为 16 轮。轮函数是扩展的 LM 结构，左右两部分分别做与 FOX64 的 LM 结构轮函数相似的变换，中间用一个线性层联系起来，如图 4.3 和图 4.4 所示。混淆函数 mu8 采用的乘法矩阵分支数为 9，S 盒与 FOX64 的 S 盒相同。

由于 FOX 密码算法的高度混淆作用和密钥编排的高度非线性型，相关密钥差分攻击对 FOX 密码不易有效 [68]，因此这里只讨论抗单密钥差分分析的安全性，不再介绍密钥编排算法。

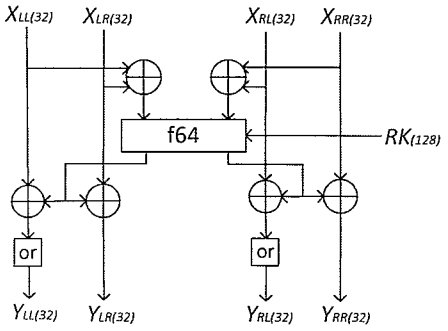


图 4.3: FOX128 轮函数

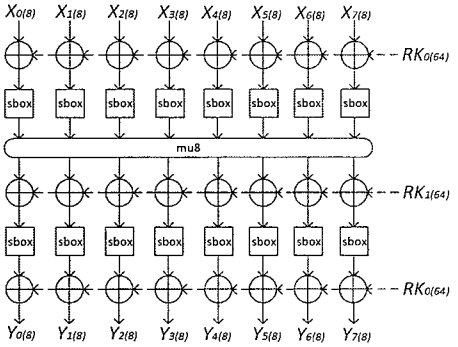


图 4.4: FOX128 中函数 f64

4.1.2 对 FOX 密码算法的已知攻击

FOX 密码算法的设计者分析了该算法在抵御多种攻击上的安全性强度，包括线性攻击、差分攻击、积分攻击 [68]。吴文玲等给出了对 4-7 轮的 FOX64 和 4-5 轮的 FOX128 的改进的积分攻击 [135]。Nakahara 等对 2 轮 FOX 做了密钥恢复攻击，对 5 轮 FOX 做了不可能差分攻击 [94]。吴文玲等后来又利用 4 轮的不可能差分区分器攻击了 5-7 轮的 FOX64 和 5 轮的 FOX128 [136]。Li 等给出了对 FOX64 的需要更少差错注入的差错攻击 [83]。Isobe 等提出了全子密钥攻击方法 [62]，攻击了 5 轮的 FOX128；后来他们将该方法改进，并用于攻击 6-7 轮的 FOX64 和 FOX128 。

尽管有上述的多种分析，但目前只有下面的由设计者本身给出的结果是对 FOX 抵抗单密钥差分攻击安全性的理论方面的证明：

定理 4.1 ([68]). *FOX64/k/r 的单密钥差分迹的差分概率上界为* $(\mathrm{DP}_{\max}^{\mathrm{sbox}})^{2r}$，*FOX128/$k$/$r$ 的单密钥差分迹的差分概率上界为* $(\mathrm{DP}_{\max}^{\mathrm{sbox}})^{4r}$。

4.2 基于 MILP 的差分分析方法

本节给出基于 MILP 的基本差分分析方法及其在 FOX 密码分析中的应用。正是由于原方法在 FOX 密码分析中的应用结果并不令人满意，才促使我们对其建模方法进行改进。

4.2.1　基于 MILP 的计算活跃 S 盒个数的方法

设一个字级的分组密码由以下三种操作组成：

- 异或操作 $\oplus$：$\mathbb{F}_2^\omega \times \mathbb{F}_2^\omega \to \mathbb{F}_2^\omega$

- S 盒替换 S：$\mathbb{F}_2^\omega \to \mathbb{F}_2^\omega$

- 线性变换 L：$\mathbb{F}_{2^\omega}^m \to \mathbb{F}_{2^\omega}^m$，其分支数为

$$\mathcal{B}_L = \min_{a \neq 0}\{w(a||L(a)) : a \in \mathbb{F}_{2^\omega}^m\},$$

其中 $w(x)$ 为 $\mathbb{F}_{2^\omega}$ 上的 $2m$-维向量 x 的非 0 元素的个数。

计算由以上三种操作构成的分组密码的最小活跃 S 盒个数的问题可以转化为线性规划问题，差分在这些操作上的传播可以用线性不等式或等式描述。用二元变量代表相应位置的差分活跃性，有 0、1 两种取值。在 Mouha 等的方法中，变量代表字级差分，值为 1 代表着该字上的差分非 0，值为 0 代表该字上的差分为 0；在 Sun 等的方法中，变量是比特级的差分。

描述异或操作的约束. 异或操作的差分分支数是 2，令输入差分为 $(\Delta a, \Delta b)$，输出差分为 Δc，引入虚变量 $d_\ominus$，描述差分在异或操作上传播的不等式为：

$$\begin{aligned} \Delta a + \Delta b + \Delta c &\geq 2d_\oplus, \\ d_\oplus &\geq \Delta a, \\ d_\oplus &\geq \Delta b, \\ d_\oplus &\geq \Delta c. \end{aligned} \tag{4.1}$$

描述线性变换的约束. 设线性变换 L 的差分分支数为 $\mathcal{B}_L$，令输入差分为 $(\Delta x_{\text{in}_0}$，
$\Delta x_{\text{in}_1}, \cdots, \Delta x_{\text{in}_{m-1}})$，输出差分为 $(\Delta x_{\text{out}_0}, \Delta x_{\text{out}_1}, \cdots, \Delta x_{\text{out}_{m-1}})$，引入虚

变量 d_L，描述差分在线性变换 L 上传播的线性不等式为：

$$\begin{aligned} &\sum_{i=0}^{m-1}(\Delta x_{\text{in}_i}+\Delta x_{\text{out}_i}) \geq \mathcal{B}_L d_L, \\ &d_L \geq \Delta x_{\text{in}_i}, i \in \{0,1,\cdots,m-1\}, \\ &d_L \geq \Delta x_{\text{out}_i}, i \in \{0,1,\cdots,m-1\}. \end{aligned} \tag{4.2}$$

在 Mouha 等的方法中，上述变量都是字级的，而在 Sun 等的方法中，上述变量都是比特级的。对于字级约束，无需对 S 盒进行额外的描述，因为 S 盒的输入差分与输出差分活跃性是一致的，代表该位置差分的字级变量本身即可表示 S 盒的活跃性。而比特级的约束需要如下对 S 盒的描述。

描述 S 盒的约束. 令 S 盒的比特级输入差分和输出差分分别为 $(\Delta y_{\text{in}_0},\cdots,\Delta y_{\text{in}_{\omega-1}})$ 和 $(\Delta y_{\text{out}_0},\cdots,\Delta y_{\text{out}_{\omega-1}})$，引入比特变量 A_t 来表示该 S 盒的活跃性，满足当至少一个输入差分比特非 0 时，$A_t=1$，当所有输入比特都为 0 时，$A_t=0$。可以统一为：

$$\begin{cases} A_t-\Delta y_{\text{in}_i} \geq 0, i \in \{0,\cdots,\omega-1\} \\ \Delta y_{\text{in}_0}+\cdots+\Delta y_{\text{in}_{\omega-1}}-A_t \geq 0. \end{cases} \tag{4.3}$$

此外，非 0 输入差分向量一定要对应非 0 的输出差分向量，反之亦然。即应满足：

$$\begin{cases} \omega\Delta y_{\text{out}_0}+\cdots+\omega\Delta y_{\text{out}_{\omega-1}}-(\Delta y_{\text{in}_0}+\cdots+\Delta y_{\text{in}_{\omega-1}}) \geq 0, \\ \omega\Delta y_{\text{in}_0}+\cdots+\omega\Delta y_{\text{in}_{\omega-1}}-(\Delta y_{\text{out}_0}+\cdots+\Delta y_{\text{out}_{\omega-1}}) \geq 0. \end{cases} \tag{4.4}$$

文献 [119] 中对 S 盒的约束有进一步的优化，但是由于这种优化只适合于轻量级 S 盒，对 FOX 这种非轻量级的 S 盒并不适用，所以不予考虑。

目标函数. MILP 问题的目标函数即是所有表示 S 盒活跃性变量的和。最小化这个目标函数即可给出活跃 S 盒个数的最小值。

另外要添加保证至少有一个 S 盒活跃的约束，防止全零的平凡情况出现。

关于用字级和比特级的 MILP 方法计算活跃 S 盒个数的详细描述可以参考文献 [93] 和 [117, 119]。

4.2.2　原方法产生的无效差分迹

将原始的 MILP 方法应用于 FOX 密码算法时，不管对多少轮进行建模，得到的 MILP 问题的最优解总是不变的，显然这样得到的活跃 S 盒个数的下界也是过于松弛的。更为严重的是，由现有的方法导出的差分迹是无效的差分迹。

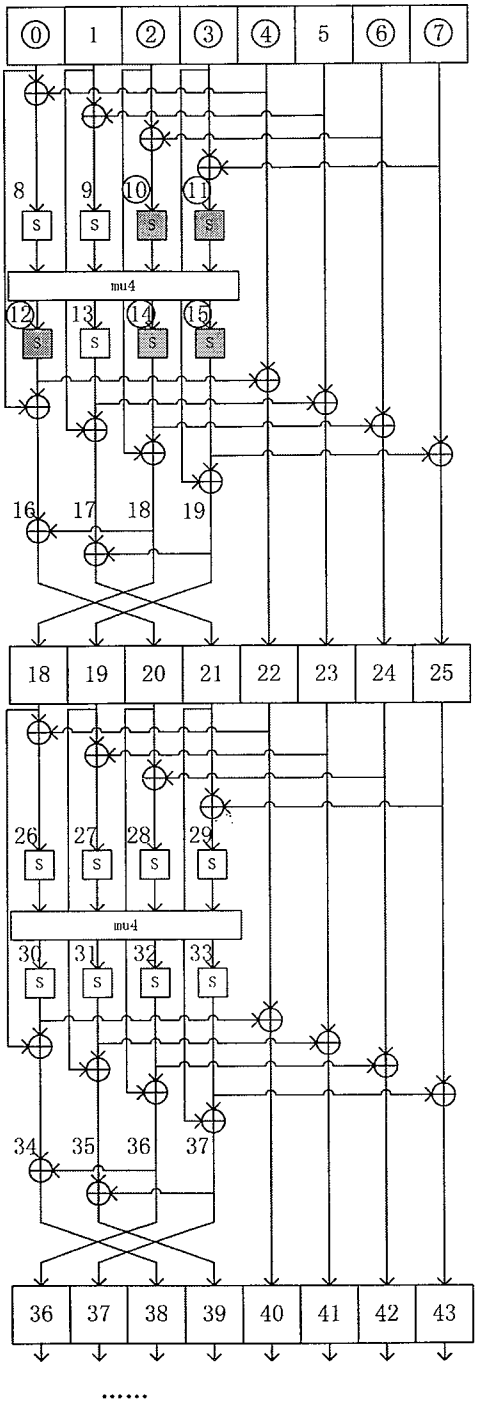
考虑方程 (4.1) 中对差分在异或操作上传播的约束，允许的差分传播形式有以下几种：

$$\begin{aligned}
&(1,1) \to 1, (d_{\oplus} = 1)\\
&(1,1) \to 0, (d_{\oplus} = 1)\\
&(1,0) \to 1, (d_{\oplus} = 1)\\
&(0,1) \to 1, (d_{\oplus} = 1)\\
&(0,0) \to 0, (d_{\oplus} = 0).
\end{aligned} \tag{4.5}$$

当考虑字级的模型时，(4.5) 中的差分传播形式都是可行的。但是当考虑比特级的模型时，(4.5) 中的第一个传播模式显然是错误的，这就使得线性规划问题在更大范围的可行域内求解，所得目标函数的最优解过于松弛，进而导致了 MILP 方法在 FOX 密码算法上的实效。

我们用简单的程序来产生 MILP 问题的算例，然后用优化问题求解软件 Gurobi 5.6 [59] 来求解。对单密钥下的 FOX 的 1-16 轮进行原始方法的建模和求解，FOX64 的目标函数个数都是 5，FOX128 的都是 9。图 4.5 展示了求解器给出的 FOX64 的最优差分迹的前两轮，图中的数字是对字级变量的编号，圆圈表示该位置的差分非 0，无圆圈表示该位置差分为 0。令 x_i 表示编号为 $i(i = 0, 1, 2, \cdots)$ 的字级差分值，$\underline{x}_i$ 表示差分 x_i 经 S 盒后的输出差分值，带阴影的 S 盒是活跃 S 盒。在该差分迹中存在矛盾，一方面有 $x_2 \oplus x_6 = x_{10} \neq 0$，对应着一个活跃 S 盒，有 $x_2 \neq x_6$ 成立；另一方面，有 $x_{18} = \underline{x}_{14} \oplus x_2 = 0$ 和 $x_{24} = \underline{x}_{14} \oplus x_6 = 0$，得到 $x_2 = x_6$ 成立，产生矛盾。类似的矛盾在 FOX128 中也存在，见附录 B。

当把密码算法本身的结构考虑在内时这种矛盾显露出来了，但是这些字级的差分赋值并不违反 Mouha 等的 MILP 方法中的约束。主要原因是方程 (4.5) 中对异或操作的约束只考虑了分支数，而没有考虑差分比特间的具体运算。以



圆圈位置差分非 0，无圆圈位置差分为 0，$\underline{x}_i$ 表示差分 x_i 经 S 盒的输出差分值，阴影 S 盒活跃。一方面有 $x_2 \oplus x_6 = x_{10} \neq 0$，对应着一个活跃 S 盒，有 $x_2 \neq x_6$ 成立；另一方面，有 $x_{18} = \underline{x}_{14} \oplus x_2 = 0$ 和 $x_{24} = \underline{x}_{14} \oplus x_6 = 0$，得到 $x_2 = x_6$ 成立，产生矛盾。

图 4.5: FOX64 无效差分迹

这种方式对字级差分的赋值是建立在各个操作相互独立的假设上的，而各操作之间的相关性在计算差分迹的概率时应当被考虑在内 [92]。虽然例子中的模型是利用的 Mouha 等的字级的 MILP 方法进行试验的，但是 Sun 等的比特级方法依然会得到这样的结果。只要把当前每一个字级的差分分割成多个相同的比特级差分即可转换成比特级的模型解，因此 Sun 等的方法依然允许这类无效差分的存在。

4.3　改进的 MILP 方法及其在 FOX 密码上的应用

本节，我们用改进的线性约束来精确刻画差分在异或操作上的传播，从而改进 Mouha 等和 Sun 等的基于 MILP 技术的方法，并应用于 FOX 密码算法。

4.3.1　改进的 MILP 建模方法

改进的基于 MILP 的方法将字级和比特级的建模方法相结合，在描述异或操作上的差分传播时采用新的比特级约束，在描述线性变换层上的差分传播时采用字级约束，在描述 S 盒层上的差分传播时采用 Sun 等的方法 [117] 实现比特级变量和字级变量之间的转换。

为了用实数域上的线性等式和不等式来表示比特异或运算，我们用文献 [32] 中的线性化技术。具体地，用式 (4.6) 中的约束代替式 (4.1) 中的约束，其中变量 $\Delta a, \Delta b, \Delta c$ 代表比特差分，$d_{\oplus}$ 依然是虚变量。

$$\begin{aligned} &\Delta c = \Delta a + \Delta b - 2d_{\oplus}, \\ &d_{\oplus} \leq \Delta a, \\ &d_{\oplus} \leq \Delta b, \\ &d_{\oplus} \geq \Delta a + \Delta b - 1, \\ &d_{\oplus} \geq 0. \end{aligned} \tag{4.6}$$

式 (4.6) 中的约束允许的可行差分传播方式为

$$\begin{aligned}(1,1) &\to 0, (d_{\oplus}=1)\\ (1,0) &\to 1, (d_{\oplus}=0)\\ (0,1) &\to 1, (d_{\oplus}=0)\\ (0,0) &\to 0, (d_{\oplus}=0)\end{aligned} \tag{4.7}$$

与比特级差分的异或运算规则完全一致，使得线性规划问题的可行域更小，从而得到更紧的下界，消除了 4.2.2 节中的无效差分迹。相对于原来字级的建模方法，新的方法使用了更多的变量和约束，问题规模有所增大。相对于原来比特级的建模方法，新的方法在问题规模上并没有显著的增大。

注意到在 FOX 算法中的 f32 和 f64 函数中，第一层 S 盒紧邻着线性变换层，紧接着是第二层 S 盒，因此对第一层 S 盒差分传播的约束可以不包括式 (4.4)，只保留式 (4.3) 中的约束，其中表示 S 盒活跃性的变量可以直接作为字级差分变量用于对线性层差分传播的约束。同理，对第二层 S 盒的约束也可以仅包含式 (4.3) 中的约束，只是需要将输入差分比特替换为输出差分比特来定义其活跃性。

4.3.2 对 FOX 密码的分析结果

我们将改进后的方法仍然应用在 FOX 密码算法上，用程序产生 MILP 的算例，即一个“lp”格式的文档，然后调用 Gurobi 5.6 求解。对 r 轮的 FOX64，算例含 $(256r+64)$ 个变量和 $(641r+1)$ 个约束；对 r 轮的 FOX128，算例中含 $(529r+128)$ 个变量和 $(1281r+1)$ 个约束。求解过程在 32 核 AMD 处理器 6132 HE @ 2.2 GHz 上进行，1-5 轮的实验可以在数秒内完成，6 轮的 FOX64 需数小时，6 轮的 FOX128 实验因内存不足终止。实验数据和得到的差分活跃 S 盒个数下界在表 4.1 中给出。

由差分迹中的活跃 S 盒的个数 n 可以算出差分迹的概率为 $(\mathrm{DP}_{\max}^{\mathrm{sbox}})^n$，反之，根据定理 4.1，$r$ 轮的 FOX64 和FOX128 的等效活跃 S 盒个数下界分别为 $2r$ 和 $4r$。用我们的方法得到的差分活跃 S 盒个数下界与定理 4.1 中的结果的比较在表 4.2 中给出。从实验结果可以看出，6 轮的 FOX64 有至少 16 个差分活跃 S 盒，差分迹的概率最大为 $(\mathrm{DP}_{\max}^{\mathrm{sbox}})^{16}=2^{-64}$，考虑到分组长度为 64 比特，因此可以抵御基本的差分攻击；12 轮的 FOX64 差分迹的最大概率为 2^{-128}，考虑到

表 4.1: 改进的 MILP 方法在 FOX 上的实验数据和结果

轮数	FOX64				FOX128			
	变量数	约束数	时间(s)	活跃 S 盒个数下界	变量数	约束数	时间(s)	活跃 S 盒个数下界
1	329	642	0.02	5	657	1282	0.00	9
2	594	1283	0.99	5	1186	2563	2.17	9
3	859	1924	44.43	6	1715	3844	134.38	10
4	1124	2565	6.11	10	2244	5125	31.88	18
5	1389	3206	6397.65	12	2773	6406	4183.49	20
6	1654	3847	25701.33	16	3302	7687	-	-

表 4.2: FOX 差分活跃 S 盒个数下界比较

轮数	FOX64		FOX128	
	文献 [68]	本章	文献 [68]	本章
1	2	5	4	9
2	4	5	8	9
3	6	6	12	10
4	8	10	16	18
5	10	12	20	20
6	12	16	24	-

密钥长度为 128 比特，因此可以抵抗密钥恢复攻击。我们的方法得到的活跃 S 盒个数下界比前人的结果更紧。对 FOX128, 实验结果表明 4 轮的 FOX128 有至少 18 个差分活跃 S 盒，则全 16 轮的 FOX128 含有至少 72 个，足够抵抗基本的单密钥差分攻击。

4.4　小结

本章，我们改进了 Mouha 等和 Sun 等的基于 MILP 技术求解差分活跃 S 盒个数下界的方法，主要的改进点在于用新的线性不等式对差分在异或操作上的传播进行了约束。新的约束精确刻画了差分比特的异或运算，合理缩小了线性规划问题的可行域，消除了一类无效的差分迹模式。将改进后的方法应用于 FOX 分组密码算法上，得到了在单密钥模型下抵抗基本差分分析的更好的安全性估计，证明了 6 轮的 FOX64 的最大差分迹概率为 2^{-64}，12 轮的 FOX64 可以抵抗密钥恢复攻击。这是目前已知对 FOX64 在单密钥模型下抗差分分析安全性的最好估计。对 FOX128 的实验结果与前人相比提升并不明显，但仍然证明了

全轮 FOX128 可抵抗基本差分分析。改进后的方法可以应用于其他由 S 盒替换、线性变换和异或运算构成的对称密码算法。

本章内容对应文献 [105]，感谢合作者的贡献。

第五章　自动化动态密钥猜测方法及其在 SIMON 和 Simeck 密码分析中的应用

SIMON 和 SPECK [13] 是 2013 年由美国国家安全局 (National Security Agency, NSA) 提出的两套轻量级分组密码算法，受到了密码学界的广泛关注 [2, 4, 5, 27, 110, 119, 126]。SIMON 是面向硬件实现的算法，SPECK 是面向软件的算法。在 CHES 2015 上，Yang 等 [137] 将 SIMON 和 SPECK 的优点结合，设计了一套新的分组密码算法，称为 Simeck。Simeck 算法的轮函数是 SIMON 轮函数的不同参数版本，并且仿照 SPECK 将轮函数在密钥编排中重复使用的方式，在密钥编排中再次使用了类似的轮函数。Simeck 的硬件实现表现在面积和能耗方面比 SIMON 更优 [137]。

2014 年，一种新的利用动态密钥猜测技术的差分分析方法被提出 [126]，并用于分析 SIMON 密码。该方法的基本思想是将经典的差分分析 [25] 和广泛应用于杂凑函数分析的模差分分析 [36, 82, 87, 121, 129] 相结合，主要针对以按位与作为非线性操作的密码算法。基于对差分在按位与操作上的传播特性的观察，攻击者可以求解出一些子密钥比特，从而极大地降低需要猜测的密钥空间。在文献 [126] 中，配合先前文献中的高概率差分 [2, 27, 118]，动态密钥猜测技术使得对 SIMON 分组密码的最长攻击轮数提高了 2-4 轮。

由于动态密钥猜测技术是新近提出的，Simeck 的设计者在对 Simeck 进行安全性自评估时并没有考虑动态密钥猜测技术，而是给出了多种其他安全性分析，包括差分分析 [25]、线性分析 [85]、不可能差分分析 [23] 等。因为 Simeck 和 SIMON 的相似性，对 Simeck 的分析主要遵循了对 SIMON 的分析方法。后来出现了对 Simeck 更多轮数的分析结果 [11, 77]，其中 Kölbl 等给出了高概率的差分迹，对 Simeck 的三个版本分别进行了 19 轮，26 轮和 33 轮的攻击 [77]。他们注意到了动态密钥猜测方法，但是并没有实现。

差分分析与差分区分器密切相关，覆盖轮数更多或概率更高的差分区分器能够使得差分分析的轮数增多，数据和时间复杂度降低。在 CRYPTO 2015 上，Kölbl 等 [76] 找到了 SIMON48，SIMON32 和 SIMON64 的新的约减轮差分，可以用于在轮数、时间复杂度或数据复杂度方面提高先前的差分攻击结果。

本章给出了动态密钥猜测技术的实现细节，用程序记录了动态密钥猜测过程以及复杂度的部分计算，并用于分析 Simeck 和 SIMON 算法。我们用基于 MILP 的方法找到了 13-轮 Simeck32/64 的低重量差分，概率是 $2^{-29.64}$。利用这条差分以及文献 [77] 中的差分，用动态密钥猜测技术分析了 21-22 轮的 Simeck32/64，28 轮的 Simeck48/96 和 34-35 轮的 Simeck64/128 算法。此外，对 SIMON 利用文献 [76] 中的差分进行了 22 轮 SIMON32/64，24 轮 SIMON48/96，29 轮 SIMON64/96 和 30 轮 SIMON64/128 的分析。对 Simeck 和 SIMON 的分析结果及与前人工作的对比分别在表 5.1 和表 5.2 中给出。

表 5.1: Simeck 算法分析结果及比较

版本	总轮数	攻击轮数	时间复杂度	数据复杂度	成功率	参考
Simeck32/64	32	18	$2^{63.5}$	2^{31}	47.7%	[11]
		19	2^{36}	2^{31}	-	[77]
		20	$2^{62.6}$	2^{32}	-	[137]
		20	$2^{56.65}$	2^{32}	-	[139]
		21	$2^{48.5}$	2^{30}	41.7%	**5.3.2**
		22	$2^{57.9}$	2^{32}	47.1%	**5.3.2**
		23	$2^{61.78}$	$2^{31.91}$	-	[107]
Simeck48/96	36	24	2^{94}	2^{45}	47.7%	[11]
		24	$2^{94.7}$	2^{48}	-	[137]
		24	$2^{91.6}$	2^{48}	-	[139]
		26	2^{62}	2^{47}	-	[77]
		28	$2^{68.3}$	2^{46}	46.8%	**5.3.2**
		30	$2^{92.2}$	$2^{47.66}$	-	[107]
Simeck64/128	44	25	$2^{126.6}$	2^{64}	-	[137]
		27	$2^{120.5}$	2^{61}	47.7%	[11]
		27	$2^{112.79}$	2^{64}	-	[139]
		33	2^{96}	2^{63}	-	[77]
		34	$2^{116.3}$	2^{63}	55.5%	**5.3.2**
		35	$2^{116.3}$	2^{63}	55.5%	**5.3.2**
		37	$2^{121.25}$	$2^{63.09}$	-	[107]

表 5.2: SIMON 算法分析结果及比较

版本	密钥长度	总轮数	攻击轮数	时间复杂度	数据复杂度	成功率	参考
SIMON32	64	32	21	$2^{55.25}$	2^{31}	51%	[126]
			22	$2^{58.76}$	2^{32}	31.5%	**5.4**
			23	2^{50}	$2^{30.59}$	-	[1]
SIMON48	96	36	24	$2^{87.25}$	2^{47}	48%	[126]
			24	$2^{83.10}$	$2^{47.78}$	-	[1]
			24	$2^{78.99}$	2^{48}	47.5%	**5.4**
SIMON64	96	42	28	$2^{84.25}$	2^{63}	46%	[126]
			28	$2^{75.39}$	2^{60}	50.3%	**5.4**
			29	$2^{86.94}$	2^{63}	47.5%	**5.4**
	128	44	29	$2^{116.25}$	2^{63}	46%	[126]
			29	$2^{101.40}$	2^{60}	50.3%	**5.4**
			30	$2^{110.99}$	2^{63}	47.5%	**5.4**

本章安排如下：5.1 给出了 Simeck 和 SIMON 的算法描述，5.2 给出了动态密钥猜测技术的一般性方法和实现细节。5.3 节和 5.4 节分别给出了在 Simeck 和 SIMON 上的实验结果，包括 Simeck 的新的差分区分器。5.5 节对本章进行总结。

5.1 SIMON 和 Simeck 分组密码

本节对 SIMON 和 Simeck 密码算法进行简要介绍。

5.1.1 符号说明

本章后面所用到的符号说明如下：

X^{r-1}	第 r 轮输入
L^{r-1}	X^{r-1} 的左半部分
R^{r-1}	X^{r-1} 的右半部分
K^{r-1}	第 r 轮子密钥
X_i	X 的左数第 i 个比特
$X \ggg r$	X 循环右移 r 比特
$\wedge$	比特与
%	模运算
$\cup$	集合的并
$\cap$	集合的交

5.1.2 SIMON 和 Simeck 分组密码

SIMON 和 Simeck 都采用了 Feistel 结构，不同的版本分别记作 SIMON$2n/mn$ 和 Simeck$2n/mn$，$2n$ 表示分组长度，mn 表示密钥长度，都是分块长度 n 的倍数。Simeck 有三个版本，为 Simeck32/64，Simeck48/96 和 Simeck64/128，对应的轮数分别为 $n_r = 32, 36, 44$ 轮。SIMON 共有 10 个版本，本章只关注其中 SIMON32/64，SIMON48/96，SIMON64/96 和 SIMON64/128 四个版本。SIMON 和 Simeck 的轮函数非常相似:

$$(L^i, R^i) = (R^{i-1} \ominus F(L^{i-1}) \oplus K^{i-1}, L^{i-1}),$$

其中

$$F(x) = ((x \lll a) \wedge (x \lll b)) \ominus (x \lll c),$$

对 Simeck，$a = 0, b = 5, c = 1$；在 SIMON 中，$a = 1, b = 8, c = 2$。密钥编排部分在本章的单密钥差分分析中不涉及，具体描述可参考文献 [137] 和 [13]。

5.2　动态密钥猜测方法及其自动化实现

2014 年王小云的团队提出了动态密钥猜测技术 [126]，极大地减少了差分分析中需要猜测的密钥空间。该技术基于对差分在按位与操作上传播特性的观察，按位与操作的不同输入差分会导致差分方程中涉及的密钥比特的不同情况，从某些差分方程中可以求解出子密钥比特的值。在基于动态密钥猜测技术的差分分析中，攻击者首先找到差分区分器，然后向前后扩展若干轮，在扩展的差分迹中，找到对差分区分器起决定作用的充分比特条件。然后建立这些充分比特条件上的差分方程，从差分方程中尽可能多地求解出密钥比特信息。文献 [126] 中给出了这项技术的一些原始准则。下面，我们详细给出这项技术的一般性描述以及在实现过程中的算法细节。

5.2.1　扩展路径的生成

设概率为 p 覆盖 R 轮的差分区分器已经找到，在区分器前面添加了 r_0 轮，在区分器后面添加了 r_1 轮。为了获取前面添加的 r_0 轮的差分迹，需要将差分区分器的输入差分进行“解密”，该过程需要遵循的规则是当按位与操作的输入差分为 (0,0) 时输出差分为 0，否则，按位与的输出差分置为 $*$。对于线性运算，未知输入差分比特产生的输出差分比特也是未知，同样置为 $*$。对于后面添加的 r_1 轮，按照相同的规则“加密”差分区分器的输出差分。

在扩展轮的差分迹中，需要找出从加密向获得区分器输入差分和解密向获得区分器输出差分的充分比特条件，然后在这些比特差分上建立差分方程。具体地说，若按位与操作的输入差分非 (0,0)，而输出差分是确定的，那么这个输出差分就应该标记为充分比特条件。在区分器前面添加的轮需要从加密方向上进行标记，在区分器后面添加的轮需要从解密方向上进行标记。表 5.3 给出了扩展轮差分迹的例子，充分比特条件用黑体标出。

5.2.2　数据收集

在扩展的差分迹中，设明文差分中有 l_0 个确定的差分比特，ΔX^1 中有 l_1 个充分比特条件，这 (l_0+l_1) 个位置是将明文数据进行划分的依据。具体地说，将明文分为 $2^{l_0+l_1}$ 个结构，每个结构中有 $2^{2n-l_0-l_1}$ 个明文，对应的 (l_0+l_1) 个位置上是相同常数。

表 5.3: 21-轮 Simeck32/64 的扩展差分迹

轮数	每轮输入差分
0	$1,*,0,0,0,*,*,*,0,*,*,*,*,1,*,*,*,*,*,0,*,*,*,*,*,*,*,*,*,*,*$
1	$\mathbf{0},*,\mathbf{0},0,\mathbf{0},\mathbf{0},*,\mathbf{0},\mathbf{0},\mathbf{0},*,*,*,\mathbf{0},\mathbf{1},*,1,*,0,0,0,*,*,*,0,*,*,*,*,1,*,*$
2	$0,\mathbf{1},0,0,0,\mathbf{0},\mathbf{0},\mathbf{0},0,\mathbf{0},\mathbf{0},\mathbf{1},\mathbf{0},0,\mathbf{0},\mathbf{1},0,*,0,0,0,0,*,0,0,0,*,*,*,0,1,*$
3	$1,\mathbf{0},0,0,0,0,\mathbf{0},0,0,0,\mathbf{0},\mathbf{0},\mathbf{0},0,0,0,\mathbf{0},0,1,0,0,0,0,0,0,0,0,0,1,0,0,0,1$
3→16	13-轮差分区分器
16	$0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\mathbf{0},0,0,0,0,0,0,0,0,0,0,\mathbf{0},0,0,0$
17	$1,*,0,0,0,0,0,0,0,0,0,0,*,0,0,0,\mathbf{0},\mathbf{1},0,0,0,0,0,\mathbf{0},0,0,0,\mathbf{0},\mathbf{0},0,0,0$
18	$*,*,0,0,0,0,0,*,0,0,0,*,*,0,0,1,\mathbf{1},*,\mathbf{0},0,0,0,\mathbf{0},\mathbf{0},0,0,\mathbf{0},\mathbf{0},*,0,0,\mathbf{0}$
19	$*,*,*,0,0,0,*,*,0,0,*,*,*,0,1,*,*,*,\mathbf{0},0,0,\mathbf{0},\mathbf{0},*,0,\mathbf{0},\mathbf{0},*,*,\mathbf{0},\mathbf{0},\mathbf{1}$
20	$*,*,*,0,0,*,*,*,0,*,*,*,*,*,*,*,*,*,*,0,\mathbf{0},\mathbf{0},*,*,\mathbf{0},\mathbf{0},*,*,*,\mathbf{0},\mathbf{1},*$
21	$*,*,*,0,*,*,*,*,*,*,*,*,*,*,*,*,*,*,*,\mathbf{0},\mathbf{0},*,*,*,\mathbf{0},*,*,*,*,*,*,*$

观察差分迹的上述 (l_0+l_1) 个位置中差分为 1 的位置，将两个在这些位置上取不同值的结构分为一组用于配成差分对，可有 $2^{2(2n-l_0-l_1)}$ 对明文对。将两个结构中的明文进行加密，将密文存在表中，按照扩展差分迹中密文差分为 0 的位置的值进行索引。设密文差分中 0 比特位置有 l_2 个，在构成的密文对中可留下 $2^{2(2n-l_0-l_1)-l_2}$ 个。

设总共建立了 2^t 个明文结构，可组成 2^{t-1} 个结构对，每个结构对中留下 $2^{2(2n-l_0-l_1)-l_2}$ 个密文对，共有 $2^{t-1+2(2n-l_0-l_1)-l_2}$ 个密文对。将密文解密一轮，根据差分再次筛选。设在 $\Delta X^{r_0+R+r_1-1}$ 中有 k 个差分比特条件需要满足，则平均留下 $2^{t-1+2(2n-l_0-l_1)-l_2-k}$ 个密文对，存储于表中，记为表 T 中。同时可得，平均有 $\lambda_r = 2^{t-1+2n-l_0-l_1}\cdot p$ 个正确对可以满足差分区分器。

存储于表 T 中的消息对仍然可以根据 ΔX^2 和 $\Delta X^{r_0+R+r_1-2}$ 上建立的差分方程进行筛选，因为某些消息差分会使得差分方程无解，据此筛除。这一筛选消息对的过程与差分方程的建立和密钥比特的求解相关，将在下一小结给出。

5.2.3 差分方程的建立与密钥比特的求解

在扩展的差分迹中已经标记了充分比特条件，对每一个充分比特条件，都可以建立差分方程，以加密方向为例，差分方程即为 $\Delta X^i_{j+n} = 0$ 或

$\Delta X^i_{j+n}=1, j\in[0,n-1]$，其中

$$\begin{aligned}\Delta X^i_{j+n} =&\Delta X^{i-1}_{(j+a)\%n+n}\wedge X^{i-1}_{(j+b)\%n+n}\oplus\Delta X^{i-1}_{(j+b)\%n+n}\\ &\wedge X^{i-1}_{(j+a)\%n+n}\oplus\Delta X^{i-1}_{(j+a)\%n+n}\wedge\Delta X^{i-1}_{(j+b)\%n+n}\\ &\oplus\Delta X^{i-1}_{(j+c)\%n+n}\oplus\Delta X^{i-2}_{j+n},\end{aligned}\tag{5.1}$$

且有

$$\begin{aligned}X^{i-1}_{(j+b)\%n+n} =&X^{i-2}_{(j+b+a)\%n+n}\wedge X^{i-2}_{(j+b+b)\%n+n}\\ &\oplus X^{i-2}_{(j+b+c)\%n+n}\oplus X^{i-2}_{(j+b)\%n}\oplus K^{i-2}_{(j+b)\%n},\\ X^{i-1}_{(j+a)\%n+n} =&X^{i-2}_{(j+a+a)\%n+n}\wedge X^{i-2}_{(j+a+b)\%n+n}\\ &\oplus X^{i-2}_{(j+a+c)\%n+n}\oplus X^{i-2}_{(j+a)\%n}\oplus K^{i-2}_{(j+a)\%n}.\end{aligned}\tag{5.2}$$

因此，对每一个差分方程，有两种类型的密钥比特与之相关，第一种是线性相关的密钥比特，也即差分方程的未知变量，第二种是非线性相关的密钥比特，也即要获取差分方程的参数需要猜测的密钥比特。将 $\Delta X^{i-1}_{(j+a)\%n+n}$, $\Delta X^{i-1}_{(j+b)\%n+n}$, $\Delta X^{i-1}_{(j+c)\%n+n}\oplus\Delta X^{i-2}_{j+n}$ 称为差分方程的参数。通过猜测非线性相关的子密钥比特值确定差分方程的参数，然后可以求解出或概率求解出线性相关子密钥比特的一比特信息。

当 $(\Delta X^{i-1}_{(j+a)\%n+n},\Delta X^{i-1}_{(j+b)\%n+n})=(0,0)$ 且 $\Delta X^{i-1}_{(j+c)\%n+n}\oplus\Delta X^{i-2}_{j+n}\neq\Delta X^i_{j+n}$ 时，得到的是错误恒等式。在 ΔX_2 上建立差分方程时，由于 ΔX_0 和 ΔX_1 的计算不需要密钥，可以直接得到差分方程，因此使得错误恒等式产生的明文对可以被筛除。在解密方向上，利用 $\Delta X^{r_0+R+r_1-2}$ 上的差分方程也可以进行类似的筛除。经过筛除后剩余的数据依然存放于表，称为表 T_1。

对于有效的差分方程，相关的两种密钥比特有以下三种情况：

1. 当

$$(\Delta X^{i-1}_{(j+a)\%n+n},\Delta X^{i-1}_{(j+b)\%n+n})=(1,0),$$

差分方程的变量是与 $X^{i-1}_{(j+b)\%n+n}$ 线性相关的子密钥比特，需要猜测的密钥比特是影响

$$X^{i-2}_{(j+b+a)\%n+n}, X^{i-2}_{(j+b+b)\%n+n}, X^{i-2}_{(j+b+c)\%n+n}, X^{i-2}_{(j+b)\%n}$$

的子密钥比特。

2. 当

$$(\Delta X^{i-1}_{(j+a)\%n+n}, \Delta X^{i-1}_{(j+b)\%n+n}) = (0,1),$$

差分方程的变量是与 $X^{i-1}_{(j+a)\%n+n}$ 线性相关的子密钥比特，需要猜测的密钥比特是影响

$$X^{i-2}_{(j+a+a)\%n+n}, X^{i-2}_{(j+a+b)\%n+n}, X^{i-2}_{(j+a+c)\%n+n}, X^{i-2}_{(j+a)\%n}$$

的子密钥比特。

3. 当

$$(\Delta X^{i-1}_{(j+a)\%n+n}, \Delta X^{i-1}_{(j+b)\%n+n}) = (1,1),$$

差分方程的变量是与 $X^{i-1}_{(j+b)\%n+n}$ 线性相关和与 $X^{i-1}_{(j+a)\%n+n}$ 线性相关的子密钥比特的线性组合，需要猜测的密钥比特是影响

$$X^{i-2}_{(j+b+a)\%n+n}, X^{i-2}_{(j+b+b)\%n+n}, X^{i-2}_{(j+b+c)\%n+n}, X^{i-2}_{(j+b)\%n},$$

$$X^{i-2}_{(j+a+a)\%n+n}, X^{i-2}_{(j+a+c)\%n+n}, X^{i-2}_{(j+a)\%n}$$

的密钥比特。

当然，需要猜测的子密钥比特只需要猜测一次。给定任意一个比特位置，现在用递归的算法来确定所有影响这个比特的子密钥比特 (含线性相关和非线性相关) 和与这个比特线性相关的子密钥比特，算法伪代码见算法 1。

对于区分器后面添加的 r_1 轮，从解密方向处理每个充分比特条件，解密方向确定有影响密钥比特和线性相关密钥比特的算法见附件 C。在充分比特条件上建立完差分方程后，我们得到不同参数条件下子密钥变量求解情况的表，从这个表中可以显示在密钥猜测阶段能够额外获得的密钥比特信息，从而降低猜测密钥空间。表 5.4 给出了 21 轮 Simeck32/64 分析中的第二轮示例。

可以看到一个子密钥比特是否能从一个差分方程中求解得到决定于三个差分参数比特。有些差分比特在不只一个差分方程中充当参数，因此需要将含有相同差分参数的差分方程一起处理。具体地，将含有相关联参数的差分方程分

表 5.4: 21-轮 Simeck32/64 的第 2 轮子密钥比特求解情况

轮数	差分方程	求解密钥[1]	方程参数	Pr	$\Pr^F$
2		筛除	$(\Delta X^1_{17}, \Delta X^1_{22}, \Delta X^0_{17}) = (0,0,0)$		1/8
	$\Delta X^2_{17} = 1 \Leftrightarrow$	*	$(\Delta X^1_{17}, \Delta X^1_{22}, \Delta X^0_{17}) = (0,0,1)$	1/8	
	$\Delta(X^1_{17} \wedge X^1_{22})$	K^0_1	$(\Delta X^1_{17}, \Delta X^1_{22}) = (0,1)$	1/4	
	$\oplus \Delta X^0_{17} = 1$	K^0_6	$(\Delta X^1_{17}, \Delta X^1_{22}) = (1,0)$	1/4	
		$k^0_1 \oplus K^0_6$	$(\Delta X^1_{17}, \Delta X^1_{22}) = (1,1)$	1/4	
	$\Delta X^2_{27} = 1 \Leftrightarrow$	筛除	$(\Delta X^1_{27}, \Delta X^1_{28} \oplus \Delta X^0_{27}) = (0,0)$		1/4
	$\Delta X^1_{27} \wedge X^1_{16}$	*	$(\Delta X^1_{27}, \Delta X^1_{28} \oplus \Delta X^0_{27}) = (0,1)$	1/4	
	$\oplus \Delta X^1_{28} \oplus \Delta X^0_{27} = 1$	K^0_0	$\Delta X^1_{27} = 1$	1/2	
		筛除	$(\Delta X^1_{28}, \Delta X^1_{17}, \Delta X^0_{28}) = (0,0,1)$		1/8
	$\Delta X^2_{28} = 0 \Leftrightarrow$	*	$(\Delta X^1_{28}, \Delta X^1_{17}, \Delta X^0_{28}) = (0,0,0)$	1/8	
	$\Delta(X^1_{28} \wedge X^1_{17})$	K^0_{12}	$(\Delta X^1_{28}, \Delta X^1_{17}) = (0,1)$	1/4	
	$\oplus \Delta X^0_{28} = 0$	K^0_1	$(\Delta X^1_{28}, \Delta X^1_{17}) = (1,0)$	1/4	
		$K^0_1 \oplus K^0_{12}$	$(\Delta X^1_{28}, \Delta X^1_{17}) = (1,1)$	1/4	
		筛除	$(\Delta X^1_{22}, \Delta X^1_{27}, \Delta X^0_{22}) = (0,0,1)$		1/8
	$\Delta X^2_{22} = 0 \Leftrightarrow$	*	$(\Delta X^1_{22}, \Delta X^1_{27}, \Delta X^0_{22}) = (0,0,0)$	1/8	
	$\Delta(X^1_{22} \wedge X^1_{27})$	K^0_6	$(\Delta X^1_{22}, \Delta X^1_{27}) = (0,1)$	1/4	
	$\oplus \Delta X^0_{22} = 0$	K^0_{11}	$(\Delta X^1_{22}, \Delta X^1_{27}) = (1,0)$	1/4	
		$K^0_6 \oplus K^0_{11}$	$(\Delta X^1_{22}, \Delta X^1_{27}) = (1,1)$	1/4	
	$\Delta X^2_{23} = 0 \Leftrightarrow$	筛除	$(\Delta X^1_{28}, \Delta X^0_{23}) = (0,1)$		1/4
	$\Delta X^1_{28} \wedge X^1_{23}$	*	$(\Delta X^1_{28}, \Delta X^0_{23}) = (0,0)$	1/4	
	$\oplus \Delta X^0_{23} = 0$	K^0_7	$\Delta X^1_{28} = 1$	1/2	
		筛除	$(\Delta X^1_{26}, \Delta X^1_{31}, \Delta X^1_{27} \oplus \Delta X^0_{26}) = (0,0,1)$		1/8
	$\Delta X^2_{26} = 0 \Leftrightarrow$	*	$(\Delta X^1_{26}, \Delta X^1_{31}, \Delta X^1_{27} \oplus \Delta X^0_{26}) = (0,0,0)$	1/8	
	$\Delta(X^1_{26} \wedge X^1_{31})$	K^0_{10}	$(\Delta X^1_{26}, \Delta X^1_{31}) = (0,1)$	1/4	
	$\oplus \Delta X^1_{27} \oplus \Delta X^0_{26} = 0$	K^0_{15}	$(\Delta X^1_{26}, \Delta X^1_{31}) = (1,0)$	1/4	
		$K^0_{10} \oplus K^0_{15}$	$(\Delta X^1_{26}, \Delta X^1_{31}) = (1,1)$	1/4	
	$\Delta X^2_{21} = 0 \Leftrightarrow$	筛除	$(\Delta X^1_{26}, \Delta X^1_{22} \oplus \Delta X^0_{21}) = (0,1)$		1/4
	$\Delta X^1_{26} \wedge X^1_{21}$	*	$(\Delta X^1_{26}, \Delta X^1_{22} \oplus \Delta X^0_{21}) = (0,0)$	1/4	
	$\oplus \Delta X^1_{22} \oplus \Delta X^0_{21} = 0$	K^0_5	$\Delta X^1_{26} = 1$	1/2	
	$\Delta X^2_{31} = 1 \Leftrightarrow$	筛除	$(\Delta X^1_{31}, \Delta X^0_{31}) = (0,0)$		1/4
	$\Delta X^1_{31} \wedge X^1_{20}$	*	$(\Delta X^1_{31}, \Delta X^0_{31}) = (0,1)$	1/4	
	$\oplus \Delta X^0_{31} = 1$	K^0_4	$\Delta X^1_{31} = 1$	1/2	
	$\Delta X^2_{25} = 0 \Leftrightarrow$ X^1_{25} $\oplus \Delta X^1_{26} \oplus \Delta X^0_{25} = 0$	K^0_9		1	
	$\Delta X^2_{30} = 0 \Leftrightarrow$ X^1_{19} $\oplus \Delta X^1_{31} \oplus \Delta X^0_{30} = 0$	K^0_3		1	

[1] 第三列中，* 代表该差分方程的子密钥比特变量值不能确定，子密钥比特本身或其线性组合表示该比特信息可以确定。表中加粗横线是分组线。

Algorithm 1 X_j^i 相关子密钥比特 (加密方向) 算法

```
输入: 轮数 i，比特位置 j
输出: [Influen_keys, Linear_keys]    ▷ [有影响子密钥，线性相关子密钥]
function RelatedKeys(i, j)
    Influent_keys= [ ], Linear_keys=[ ]
    if i = 0 then
        return [Influent_keys, Linear_keys]
    else
        if j < n then
            return RelatedKeys(i − 1, j + n)
        else
            [I_0, L_0]=RelatedKeys(i − 1, (j + a)%n + n)
            [I_1, L_1]=RelatedKeys(i − 1, (j + b)%n + n)
            [I_2, L_2]=RelatedKeys(i − 1, (j + c)%n + n)
            [I_3, L_3]=RelatedKeys(i − 1, j%n)
            Linear_keys=L_2 ∪ L_3 ∪ K_{j%n}^{i−1}
            Influent_keys = I_0 ∪ I_1 ∪ I_2 ∪ I_3 ∪ K_{j%n}^{i−1}
            return [Influent_keys, Linear_keys]
        end if
    end if
end function
```

为一组，计算子密钥比特在这一组上的平均取值个数。在每一轮，设差分方程的顺序按充分比特条件的顺序排放在 $Index\_order$ 中，对应的参数集合存放在 $Para\_sets$ 中，用算法 2 对充分比特条件 (也即差分方程) 进行分组。

在实际的攻击中，每一轮首先猜测确定差分方程需要猜测的子密钥比特，然后逐组根据差分参数得到子密钥比特变量的值。在第 j 组中，设为确定差分方程猜测了 g_j 子密钥比特的值，差分方程包含 k_j 个子密钥比特变量，差分参数有 $t_{j,i}$ 种情况可以得到 $v_{j,i}$ 种子密钥变量的取值，那么这一组中的 $(g_j + k_j)$ 个子密钥比特的平均取值个数为 $2^{g_j} \cdot \frac{\sum_i t_{j,i} v_{j,i}}{\sum_i t_{j,i}}$。对一轮的所有分组，得到 $\sum_j (g_j + k_j)$ 个密钥比特的 $\prod_j (2^{g_j} \cdot \frac{\sum_i t_{j,i} v_{j,i}}{\sum_i t_{j,i}})$ 个取值。如果扩展轮中涉及的子密钥比特 (包含需要猜测的比特和变量比特) 个数少于主密钥长度，则可以以低于穷搜的复杂度进行密钥恢复攻击。

当把所有分组中的子密钥比特取值个数综合时，有两种形式的重复。第一种发生于子密钥比特在多于一组的差分方程中都是变量的情况，此时将重复的

Algorithm 2 充分比特条件分组算法

```
1: procedure GROUP(Index_order, Para_sets)
2:     Assert length(Index_order)=length(Para_sets)
3:     k=0
4:     while k <length(Index_order) do
5:         flag=0
6:         j=k+1
7:         while j <length(Index_order) do
8:             if Para_sets[j] ∩ Para_sets[k] is not empty then
9:                 Index_order[k]=Index_order[k]∪ Index_order[j]
10:                Remove Index_order[j] from Index_order
11:                Para_sets[k] = Para_sets[k]∪ Para_sets[j]
12:                Remove Para_sets[j] from Para_sets
13:                flag=1
14:            else
15:                j++
16:            end if
17:        end while
18:        if flag=0 then
19:            k++
20:        end if
21:    end while
22: end procedure
```

比特去掉。第二种发生于一些子密钥比特的线性组合作为某个差分方程的变量已经求解过一次了，之后组合中的每一个密钥比特又作为变量被求解一次，此时有一个比特的重复。对于第二种情况，进行实际攻击时，所有重复比特的值都应该保存下来，因为从有些差分方程中并不能唯一确定取值，是否重复需要根据实际情况判断。而在计算复杂度时重复比特并不产生影响，因为每有一个重复比特出现时，其取值个数也会相应翻倍，计算出的平均取值个数不变。

5.2.4　复杂度计算

给定高概率差分区分器和前后添加的轮数，我们的程序可以给出添加轮数中涉及的子密钥比特数 $|sk|$ 和这些密钥比特的平均取值个数 C_s。一个错误的猜测能被留下的概率是 $p_e = \frac{C_s}{2^{|sk|}}$，获得 T_1 中的所有明文对的计数次数平均为 $\lambda_e = N_r \times p_e$。对计数次数大于 $s = \lfloor \lambda_r \rfloor$ 的密钥值进行穷搜，然后对未涉及到

的密钥比特进行穷搜，综合两部分的复杂度，得到攻击的总时间复杂度为

$$T_{es} = 2^{mn} \times (1 - \text{Poisscdf}(s, \lambda_e)), \tag{5.3}$$

其中 $\text{Poisscdf}(\cdot, y)$ 是期望为 y 的泊松分布的累积函数。成功概率为

$$1 - \text{Poisscdf}(s, \lambda_r), \tag{5.4}$$

其中 $\text{Poisscdf}(s, \lambda_r)$ 是没有密钥比特得到 s 次以上计数次数的概率。

5.3 Simeck 密码的分析结果

5.3.1 Simeck 高概率差分闭包搜索

利用基于 MILP 的自动化差分搜索方法 [105, 117–119]，我们的得到了 Simeck32/64 的一条 13-轮差分迹，概率为 2^{-38}，如表 5.5 所示。然后搜索与这条差分迹有相同输入差分和相同输出差分且概率 q 满足 $2^{-50} \leq q \leq 2^{-38}$ 的差分迹，其概率分布如表 5.6 所示。综合得到的所有差分迹，我们得到差分区分器 $(0x0, 0x2) \to (0x2, 0x0)$ 的概率是 $2^{-29.64}$。

表 5.5: 13-轮 Simeck32/64 概率为 2^{-38} 的差分迹

轮数	输入差分
0	0000000000000000 0000000000000010
1	0000000000000010 0000000000000000
2	0000000000000100 0000000000000010
3	0000000000001010 0000000000000100
4	0000000000010000 0000000000001010
5	0000000000111010 0000000000010000
6	0000000000001100 0000000000111010
7	0000000000101010 0000000000001100
8	0000000000010000 0000000000101010
9	0000000000001010 0000000000010000
10	0000000000000100 0000000000001010
11	0000000000000010 0000000000000100
12	0000000000000000 0000000000000010
13	0000000000000010 0000000000000000

表 5.6: Simeck32 13-轮差分 $(0000, 0002) \rightarrow (0002, 0000)$ 的概率分布

概率	2^{-38}	2^{-40}	2^{-41}	2^{-42}	2^{-43}	2^{-44}	2^{-45}	2^{-46}	2^{-47}	2^{-48}	2^{-49}	2^{-50}	无效[1]
条数	4	62	52	427	637	2427	4384	12477	22742	48324	62039	50411	169458

[1] 无效差分迹是由于按位与操作的输入的非独立性导致的 [27, 118, 119]。

5.3.2 Simeck 实验结果

我们利用基于 MILP 技术找到的高概率差分区分器和文献 [77] 中提供的高概率差分区分器对 Simeck 进行抗动态密钥猜测差分分析的安全性评估。程序的输出提供了动态密钥猜测过程中与每个充分比特条件相关的子密钥比特信息。以利用了 13-轮差分区分器 $(0x8000, 0x4011) \rightarrow (0x4000, 0x0)$ 的 21 轮 Simeck32/64 的攻击为例，程序将区分器前添加的 3 轮和后面添加的 5 轮的信息分开给出。首先是添加轮数的差分迹，其中的充分比特条件用黑体标出 (如表 5.3 所示)，然后是从第二轮到区分器前一轮的按分组给出的充分比特条件的下标以及每个分组中的差分参数比特：

```
[[{17, 27, 28, 22, 23}, {26, 21, 31}, {25}, {30}], [{17}, {22}, {26}, {27}, {28}, {31}]]
[[{'\\Delta x^1_17', '\\Delta x^1_28', '\\Delta x^1_28 xor \\Delta x^0_27', '\\Delta x^0_22',
'\\Delta x^1_22', '\\Delta x^0_28', '\\Delta x^1_27', '\\Delta x^0_17', '\\Delta x^0_23'},
{'\\Delta x^1_22 xor \\Delta x^0_21', '\\Delta x^1_27 xor \\Delta x^0_26',
'\\Delta x^1_31', '\\Delta x^0_31', '\\Delta x^1_26'},
{'\\Delta x^1_26 xor \\Delta x^0_25'}, {'\\Delta x^1_31 xor \\Delta x^0_30'}],
[{'\\Delta x^1_17'}, {'\\Delta x^1_22'}, {'\\Delta x^1_26'}, {'\\Delta x^1_27'},
{'\\Delta x^1_28'}, {'\\Delta x^1_31'}]]
```

从程序输出可以看出，第二轮的充分比特条件被分成了四组，第三轮的充分比特条件被分成了六组。第二轮的第一组与 9 个参数相关，第二组与 5 个参数相关，剩余的类似。

然后程序以 Python 语言中“字典” (dictionary) 的结构给出对应每个比特条件的子密钥比特信息，如：

```
{'expand': '\\Delta x^1_17&x^1_22 xor \\Delta x^1_22&x^1_17 xor
\\Delta x^1_17&\\Delta x^1_22 xor 0 xor \\Delta x^0_17 = I',
'guessed key': set(),
'related key': ['\\Delta x^1_17 not zero', {'K^0_6'}, '\\Delta x^1_22 not zero', {'K^0_1'}],
'bit condition': '\\Delta x^2_17=I'}
```

每项的含义如下：

‘expand’ 对应充分比特条件的差分方程；

'guessed key' 获取差分方程需要猜测的子密钥比特；

'related key' 不同的参数条件下能获得确定值的子密钥比特 (或子密钥比特的线性组合)；

'bit condition' 充分比特条件。

然后程序给出动态密钥猜测阶段每个分组的猜测细节，包括该分组的子密钥比特变量、产生无效差分方程的参数模式、参数、子密钥比特解的个数和产生它的参数情况数、所有的解的个数、所有的情况数和有效的情况数。下面是第二轮第一个分组的例子：

```
Group  1
['K^0_15', 'K^0_10', 'K^0_5', 'K^0_4']
['*10*0', '1***0', '**00*']
{'\\Delta x^1_22 xor \\Delta x^0_21', '\\Delta x^1_27 xor \\Delta x^0_26', '\\Delta x^1_31',
'\\Delta x^0_31', '\\Delta x^1_26'}
number of key bits solutions number of conditions
0 15
16 1
2 8
4 8
all solution numbers 64
all conditions 32
valid conditions 17
```

一个分组可以得到的子密钥比特值的平均个数为该分组所有子密钥比特解的个数除以所有的参数情况数。程序在每轮最后给出获得该轮的差分方程需要猜测的子密钥比特。

```
guessed key  [].
```

然后程序列出所有差分方程的子密钥比特变量和需要猜测的子密钥比特集合。有阴影的是重复的比特，需要在计算复杂度时去除。

```
K^0_6
K^0_1
K^0_0
...
```

最后，程序给出综合所有分组后得到的子密钥比特取值个数和子密钥比特数。

```
we finally get 2^3.4729 values of 17 key bits
```

区分器后面添加的轮数也以相同的方式给出。

对 Simeck32/64 我们采用了两个差分区分器。第一个是文献 [77] 中给出的覆盖 13 轮概率为 $2^{-27.28}$ 的差分 $(0x8000, 0x4011) \rightarrow (0x4000, 0x0)$。在前面添加 3 轮在后面添加 5 轮，建立 2^{14} 个结构，每个结构中有 2^{16} 个明文，通过筛选平均剩下 $2^{31.2}$ 个明文对，正确对平均有 3.29 个。在动态密钥猜测阶段平均得到 53 个密钥比特的 $2^{19.11}$ 个取值。根据 5.2.4 节的计算，时间复杂度为 $2^{48.52}$，成功概率为 41.7%。扩展的差分迹在表 5.3 中给出。第二轮的子密钥比特在表 5.4 中给出。

由于篇幅的限制，其他版本的程序输出结果在 http://pan.baidu.com/s/1jGyBwj0 中给出。下面只给出分析的结果。对 Simeck32/64 采用的另一个差分区分器是 5.3.1 节中得到的。在该区分器的前面添加了 4 轮，后面添加了 5 轮。构造了 2^{18} 个结构，每个结构里含 2^{14} 个明文。筛选平均得到 $2^{31.9}$ 个明文对，正确对的个数平均为 2.56 个。动态密钥猜测过程平均可以得到 54 个密钥比特的 $2^{21.09}$ 个取值。时间复杂度和正确概率分别为 $2^{57.88}$ 和 47.1%。22-轮的扩展差分迹在附件中的表 C.1 给出。

对 Simeck48/96，我们使用文献 [77] 中提供的差分 $(0x400000, 0xe00000) \rightarrow (0x400000, 0x200000)$，该差分覆盖 20 轮，概率为 $2^{-43.65}$。在区分器前后各添加 4 轮，构造 2^{18} 个结构，每个结构中有 2^{28} 个明文，经数据筛选平均留下 $2^{50.46}$ 个明文对，正确对的数目平均为 2.54 个。动态密钥猜测过程平均可以得到 75 个密钥比特的 $2^{32.89}$ 个值。攻击的时间复杂度和成功概率分别为 $2^{68.31}$ 和 46.8%。28-轮 Simeck48/96 的扩展差分迹在附录的表 C.2 中给出。

对 Simeck64/128，我们使用文献 [77] 中提供的差分 $(0x0, 0x4400000) \rightarrow (0x8800000, 0x400000)$，该差分覆盖 26 轮，概率为 $2^{-60.02}$。在区分器前后各添加 4 轮，构造 2^{42} 个结构，每个结构中有 2^{21} 个明文，经数据筛选留下 $2^{38.59}$ 个明文对，正确对的个数平均为 3.94 个。动态密钥猜测阶段平均可以得到 82 个密钥比特的 $2^{41.72}$ 个取值，时间复杂度和成功概率分别为 $2^{116.27}$ 和 55.5%。如果在区分器前面多添加一轮，可以以相同的数据复杂度和时间复杂度得到 35-轮的 Simeck64/128 的攻击。不同点是选择 2^{31} 个结构，每个结构里 2^{32} 个明文，经筛选期望得到 $2^{49.05}$ 对明文对。在密钥猜测阶段平均得到 118 个密钥比特的 $2^{67.26}$ 个取值。35-轮 Simeck64/128 的扩展差分迹在附录的表 C.3 给出。

对 Simeck 的差分分析数据在表 5.7 中给出。

表 5.7: Simeck 的差分分析

版本	攻击轮数	$\|sk\|$	λ_e	λ_r	数据复杂度	时间复杂度	成功概率
Simeck32	21	53	$2^{-2.678}$	3.29	2^{30}	$2^{48.52}$	41.7%
	22	54	2^{-1}	2.56	2^{32}	$2^{57.88}$	47.1%
Simeck48	28	75	$2^{-8.365}$	2.54	2^{46}	$2^{68.31}$	46.8%
Simeck64	34	82	$2^{-1.678}$	3.94	2^{63}	$2^{116.34}$	55.5%
	35	118	$2^{-1.678}$	3.94	2^{63}	$2^{116.34}$	55.5%

5.4 SIMON 密码的分析结果

对 SIMON 的分析结果主要基于 Kölbl 等找到的新的高概率差分 [76]。分析结果在表 5.8 中给出。由于篇幅的限制，我们将程序输出在 http://pan.baidu.com/s/1jGyBwj0 中给出，下面只对基本信息进行说明。

对 SIMON32/64，在覆盖 14 轮、概率为 $2^{-30.81}$ 的差分 $(0x0, 0x8) \rightarrow (0x800, 0x0)$ 的前面添加 3 轮，后面添加 5 轮。构造 2^{25} 个结构，每个含有 2^7 个明文，经过筛选平均得到 $2^{43.58}$ 个明文对，其中平均有 1.14 个正确对。动态密钥猜测阶段平均得到 55 个密钥比特的 $2^{26.33}$ 个取值。

对 SIMON48/96，在覆盖 17 轮、概率为 $2^{-46.32}$ 的差分 $(0x80, 0x222) \rightarrow (0x222, 0x80)$ 前面扩展 3 轮，在后面扩展 5 轮。构造 2^{30} 个结构，每个结构中有 2^{18} 个明文。经过筛选，平均得到 $2^{45.43}$ 个明文对，正确对的期望个数为 1.6。动态密钥猜测阶段得到 79 个密钥比特的 $2^{25.56}$ 个取值。

对 SIMON64，在覆盖 21 轮、概率为 $2^{-57.57}$ 的差分 $(0x4000000, 0x11000000) \rightarrow (0x11000000, 0x4000000)$ 的前面扩展 3 轮，后面扩展 4 轮，可以进行 28 轮的攻击。建立 2^{48} 个结构，每个结构里有明文 2^{12} 个，经筛选得到 $2^{30.49}$ 个明文对，平均有 2.69 个正确对。动态密钥猜测阶段平均得到 74 个子密钥比特的 $2^{37.5}$ 个值。如果在区分器前多扩展一轮，通过建立每个结构中含 2^{27} 个明文的 2^{33} 个明文结构，可以进行 29 轮攻击，在密钥猜测阶段平均得到 106 个比特的 $2^{53.35}$ 个值。这两个结果在时间复杂度和数据复杂度上比之前最好的结果 [126] 更好。

另外一条覆盖 22 轮、概率为 $2^{-61.32}$ 的差分路径 $(0x440, 0x1880) \rightarrow (0x440, 0x100)$ 可以用来攻击更多轮数的 SIMON64/96 和 SIMON64/128。在区分器的前面扩展 3 轮、后面扩展 4 轮，建立 2^{41} 个结构，每个结构中含 2^{22} 个明文，可以用来攻击 29 轮 SIMON64/96。动态密钥猜测阶段平均得到 84 个密钥比特的 $2^{37.41}$ 个取值。在区分器前多扩展 1 轮，可以攻击 30 轮 SIMON64/128。建立 2^{22} 个结构，每个结构含 2^{41} 个明文。在密钥猜测阶段平均得到 118 个比特的 $2^{50.77}$ 个取值。这两个结果比之前对 SIMON64 最好的分析结果 [126] 多攻击 1 轮。

22 轮 SIMON32，24 轮 SIMON48 和 29、30 轮 SIMON64 的扩展差分迹在附件的表 C.4, C.5, C.6 和 C.7 中给出。

表 5.8: SIMON 的差分分析

版本	攻击轮数	$\|sk\|$	λ_e	λ_r	数据复杂度	时间复杂度	成功概率
SIMON32/64	22	55	2^{-2}	1.14	2^{32}	$2^{58.76}$	31.5%
SIMON48/96	24	79	2^{-8}	1.6	2^{48}	$2^{78.99}$	47.5%
SIMON64/96	28	74	2^{-6}	2.69	2^{60}	$2^{75.39}$	50.3%
	29	84	2^{-4}	1.6	2^{63}	$2^{86.94}$	47.5%
SIMON64/128	29	106	2^{-8}	2.69	2^{60}	$2^{101.4}$	50.3%
	30	118	2^{-8}	1.6	2^{63}	$2^{110.99}$	47.5%

5.5　小结

本章，我们把动态密钥猜测方法应用到 CHES 2015 上提出的轻量级分组密码算法 Simeck 以及四个版本的 SIMON 上，给出了新的分析结果。我们利用基于 MILP 的方法找到了 Simeck32 的 13-轮低重量、高概率的差分，利用该差分能获得更好的攻击结果。我们用程序实现了动态密钥猜测技术，程序可以自动化给出 SIMON 和 Simeck 类型的分组密码在差分分析中的扩展差分路径和动态密钥猜测阶段的密钥比特信息，从而更方便地估计差分安全性。对 Simeck 密码，我们给出了 21、22-轮的 Simeck32，28 轮的 Simeck48，34、35 轮的 Simeck64 的分析结果。对 SIMON 密码，我们给出了 22 轮的 SIMON32，24 轮的 SIMON48，28、29 轮的 SIMON64/96 和 29、30 轮的 SIMON64/128 的分析结果。其中对 SIMON64 的结果是目前最优的。

本章内容对应文献 [103]，感谢合作者的贡献。

第六章　4-比特 S 盒的低延迟实现

S 盒是对称密码算法中最为常用的非线性部件之一。对 S 盒的研究一方面有基于密码学性质如代数免疫度、抗差分安全性、抗线性安全性的分析和分类研究，一方面有对其高效率硬件实现的研究。随着轻量级密码算法的出现，对 4-比特 S 盒的研究也不断涌现。文献 [101, 122] 的作者探究了用基本操作 AND、OR、XOR、NOT、NAND、NOR、XNOR 来实现 4-比特 S 盒的方法，他们给出启发式的方法来以最小的操作序列长度实现 S 盒。文献 [116] 用基于 SAT 求解器的工具寻找 S 盒的低深度实现，同时考虑其他的实现准则。在文献 [12] 中，也用到了评估 S 盒实现深度的工具，但是并没有给出该工具的设计。本章，我们设计了一种能给出 4-比特 S 盒最小深度实现方式的工具，同时也就给出了 S 盒的最小实现深度的评估。另一方面，利用该工具，可以给出实现深度最小且具有最优差分、线性性质的 4-比特 S 盒，这些 S 盒可以用于对称密码算法尤其是轻量级对称密码算法的设计。

6.1　4-比特布尔函数的低延迟实现算法

首先，我们沿用文献 [12] 中对各基本操作实现深度的假设，如表 6.1 所示。

表 6.1: 基本操作的实现深度和面积

操作	实现深度和面积 (GE)
XOR/XNOR	2
AND/OR	1.5
NAND/NOR	1
NOT	0.5

例如，一个如下实现的布尔函数的实现深度为 3.5。

$$((c \text{ NAND } d) \text{ NAND } (b \text{ NAND } (\text{NOT } a))) \text{ NOR } (b \text{ NOR } (a \text{ NAND } d)).$$

与文献 [12] 中相同，我们依旧将 4-比特 S 盒的每个输出比特看作相互独立的输出，即每个输出比特都是输入比特的布尔函数，然后对每个布尔函数给出一个最小深度的实现。最小深度也就意味着最低延迟、最高效率。整个 S 盒的最小实现深度由实现深度最大的布尔函数决定。具体地，用一个递归算法枚举出所有能在某个深度阈值内实现的 4-比特布尔函数。对每个布尔函数，记录了下面四种类型的信息:

- 真值表
- 实现表达式
- 实现深度
- 实现面积 (GE)

例如，1010101010101010 代表了布尔函数 "NOT d" 的真值表，其中 "d" 即为 S 盒的 4-比特输入 $a|b|c|d$ 的最低比特。实现深度和实现面积都是 0.5，那么关于这个布尔函数的记录就是 [1010101010101010, "NOT d", 0.5, 0.5]。枚举固定深度可实现的布尔函数的递归算法如下：

深度 0： 四个输入比特本身即四个 0-深度的布尔函数。

深度 0.5： 将基本操作 NOT 作用在所有 0-深度的布尔函数上。

深度 1： 将基本操作 NAND/NOR 作用在两个不同的 0-深度布尔函数上。

深度 1.5： 有 3 种产生实现深度为 1.5 的布尔函数的方式：1) 将基本操作 AND/OR 作用在两个不同的 0-深度布尔函数上；2) 将基本操作 NAND/NOR 作用在两个不同的 0.5-深度的布尔函数上；3) 将基本操作 NOT 作用在 1-深度的布尔函数上。

深度 $\geq$ 2: 有 4 种产生所需深度的布尔函数的方式：将基本操作 XOR/XNOR、AND/OR、NAND/NOR 和 NOT 分别作用在深度为 $d-2, d-1.5, d-1, d-0.5$ 的布尔函数上。注意对二元操作，只要有一个输入满足所需深度，另一个输入小于等于该深度即可。

对于某些可以有多种相同深度的实现方式的布尔函数，我们仅存储一个实现面积最小的实现方式。若有多种深度和面积都相同的实现方式，则存储下算法运行中第一个出现的。

4-比特 S 盒的每一个输出比特都是一个平衡的布尔函数，因此，我们只筛选出平衡布尔函数保存下来用于测试 S 盒。表 6.2 给出了可以在低深度下实现的布尔函数的个数。总个数给出了存在该深度实现方式的布尔函数个数。如果一个布尔函数可以有不同深度的实现方式，则每个深度下都有该布尔函数的计数。平衡个数为存在该深度实现方式的平衡布尔函数的个数。例如可以由深度 1 实现的布尔函数有 12 个，但是它们都不是平衡的。累积个数为以该深度为阈值可以实现的平衡布尔函数个数。如实现深度小于等于 2 的平衡布尔函数有 20 个。此时如果一个布尔函数存在多种深度的实现方式，则只计数深度最低的一次。平衡的 4-比特布尔函数共有 $\binom{16}{8} = 12870 \approx 2^{13.65}$，其中绝大部分可以在深度 4.5 内实现，仅有 64 个不在该范围内。

表 6.2: 不同实现深度下的布尔函数个数

深度	0	0.5	1	1.5	2	2.5	3	3.5	4	4.5
总个数	4	4	12	38	166	698	3320	17804	57048 ($2^{15.8}$)	65440 ($2^{15.99}$)
平衡个数	4	4	0	0	16	116	336	1684	11038 ($2^{13.43}$)	12806 ($2^{13.64}$)
累积个数	4	8	8	8	20	116	336	1684	11038 ($2^{13.43}$)	12806 ($2^{13.64}$)

6.2　4-比特 S 盒的低延迟评估及实现结果

给定一个 4-比特 S 盒，计算出每个输出比特的真值表，然后匹配存储下的布尔函数的真值表，能够得到最小深度和最小面积的实现方式。S 盒的实现深度即是深度最大的布尔函数的实现深度。由于数据的存储使用了 Python 编程语言中的字典 (dictionary) 结构，匹配过程仅需 $O(1)$ 的复杂度。这样就实现了给出任意给定 4-比特 S 盒最小深度实现方式的功能。我们将这一工具存放在 `https://github.com/qiaokexin/Sbox-depth-evaluation`，可供使用。以 `Midori` 算法 [12] 中的的 S 盒 $\mathtt{Sb}_0$ 为例，将该 S 盒的真值表作为 `find_expression` 函数的输入：

```
find_expression([0xc,0xa,0xd,0x3,0xe,0xb,0xf,0x7,0x8,0x9,0x1,0x5,0x0,0x2,0x4,0x6])
```

程序运行后可得到

```
a' = ['( a AND b ) NOR ( ( NOT c ) NOR ( a NOR d ) )', 3, 5.0]
b' = ['( ( a NOR d ) NOR ( b AND c ) ) NAND ( a NAND ( c AND d ) )', 3.5, 7.0]
c' = ['( b NOR d ) NOR ( NOT ( a NAND ( b NAND d ) ) )', 3.5, 4.5]
d' = ['( ( a NAND b ) NAND ( c OR d ) ) NOR ( a NOR ( b OR c ) )', 3.5, 7.0]
Depth is 3.5.
```

结果中给出了每个输出布尔函数的硬件实现形式、实现深度、实现面积以及 S 盒的实现深度。

另一方面，我们可以根据存储的平衡布尔函数来构造能在一定深度内实现的 4-比特 S 盒。分两步进行：

第 1 步：搜索固定深度内可以实现的 2-比特平衡的布尔函数对，即两个布尔函数的真值表中 00、01、10、11 的出现次数相同。例如，

$$\begin{pmatrix} 1100110011001100 \\ 1111111100000000 \end{pmatrix}$$

就是一个 2-比特平衡的布尔函数对。

第 2 步：在第 1 步的结果中搜索可以满足 4-比特平衡的布尔函数对的组合，即可构成一个 S 盒置换。由于只有差分性质和线性性质最优的 S 盒才在密码学中有使用价值，因此我们只保留差分性质和线性性质最优的 S 盒，即差分概率和线性偏差最大都为 2^{-2} 的 S 盒。

利用这个工具，产生了 $2^{25.2}$ 个实现深度在 3.5 以内的差分、线性性质最优 4-比特 S 盒，其中有 $2^{11.17}$ 个深度为 3 (见 https://github.com/qiaokexin/Sbox-depth-evaluation)。不存在实现深度更低的差分、线性性质最优的 4-比特 S 盒。

6.3 小结

本章，我们从布尔函数的硬件实现方式入手研制了一种关于 4-比特 S 盒硬件实现的工具，利用该工具，可以实现两方面的功能，一方面是给定 4-比特 S 盒，可以给出它的最小深度实现方式，另一方面，可以反向推选出具有最优差分性质和线性性质且能低深度实现的 4-比特 S 盒。

本章内容对应文献 [56] 中 5.1 章节内容，为本人独立完成部分，感谢合作者的贡献。

第七章　总结与展望

7.1　总结

分组密码和杂凑函数是密码学领域的重要研究对象，本文围绕分组密码和SHA-3 杂凑函数的差分分析进行研究，利用原创性的线性化子空间方法提高了SHA-3 杂凑函数的实际碰撞攻击结果，改进了基于线性规划的差分搜索方法，自动化实现了差分分析中的动态密钥猜测技术，并研制了 4-比特 S 盒低深度实现工具。研究内容和成果如下：

1. 对 SHA-3 杂凑函数标准 Keccak 系列，原创性地提出了可线性化子空间的概念和线性化方法，通过将置换函数中的 S 盒层在子空间中进行线性化的技术，扩展了代数方法与差分方法相结合的攻击方法，将 SHA-3 系列杂凑函数的实际碰撞攻击结果由之前的 4 轮提升到 5 轮，找到了 SHA-3 标准之一 SHAKE128 的 5 轮实际碰撞消息，以及两个挑战赛版本的 5 轮实际碰撞消息。

2. 对基于线性规划的自动化差分搜索方法，改进了前人在异或操作上的建模，并用在 FOX 分组密码的分析中，改进后的方法成功规避了原方法产生的无效差分，得到了对 FOX 算法更加精准的安全界估计。

3. 将适用于 Simon 和 Simeck 类型的轻量级分组密码的差分分析中的动态密钥猜测技术进行了程序实现，程序可以自动给出密钥猜测阶段的密钥比特信息以及计算复杂度，使得该类型密码差分攻击复杂度的计算更加方便。应用于 Simeck 和 Simon 密码上，结合新找到的高概率差分，进行了差分分析。对 Simon64 的分析是目前轮数最多的分析结果。

4. 研制了 4-比特 S 盒的低深度实现工具。该工具可以实现的功能是对任意给定 4-比特 S 盒，可以即时给出深度最低的硬件实现方式。利用该工具，我们还给出了具有最优差分性质和线性性质且能在最小深度内实现的 4-比特 S 盒。

7.2 展望

对分组密码和 SHA-3 杂凑函数的分析一直是密码学领域的研究热点，结合目前的发展趋势，对相关领域的发展有如下展望。

SHA-3 杂凑函数碰撞攻击方面. 目前对 SHA-3 杂凑函数的碰撞攻击还仅局限在约减轮上，即使是最优的结果 5 轮，也离全部 24 轮有较大差距，而且目前的方法并非对所有版本都适用。即使如此，目前的方法也暴露了 SHA-3 杂凑函数设计中某些不尽如人意的地方。最为明显的就是大状态的选取。状态大小为 1600 比特不管是对分组密码还是杂凑函数来说都是一个较大的数目，较大的状态对抵抗某些攻击手段是有作用的，比如基于混合整数线性规划的差分搜索方法，较大状态通常意味着较大的问题规模，从而使这些方法因为问题规模太大而无法实际进行。但是，较大的状态恰恰是本文进行 KECCAK 碰撞攻击所用方法的得益之处，因为只有较大的状态才能使得求解空间足够大，允许施加更多的约束条件，仍然能得到一个较大的解空间来从中进行碰撞搜索。较小状态的算法反而无法实施这样的攻击。由此可以看出，一味地增大状态规模并非有百利而无一害，很有可能为攻击者提供新的便利。

从具体攻击技术上讲，将代数方法与差分方法相结合的方法还有很大的发展空间。线性化 S 盒层的技术目前只用在置换函数的一轮中，只要方程组解空间自由度允许，可以进行更多轮的线性化，获得覆盖更多轮数的代数连接器。目前，以线性方程组描述若干轮，再在其解空间中搜索后续若干轮碰撞的方法还只局限在一个消息分块上，对于线性方程组解空间不够大，不足以提供充分的搜索空间的情况，可以考虑两个或多个消息分块。以两个消息分块为例，将两个消息的第一个分块置为相同的值，在第二个分块上运行连接器算法，此时需做的改动是状态的内部比特不再赋值为 0，而是赋值为第一个分块处理后对应的输出值。理论上，改变赋值对连接器的解空间大小在平均意义下并无影响。此时可以通过改变第一个分块的值来构造更多的解空间，以得到的所有解空间作为后续轮的搜索空间。

分组密码的分析方面. 本文在分析杂凑函数时所使用的线性化技术可以推广到分组密码的分析中，获得更高轮数的分析结果。当前对设计较好的分组密码的差分分析通常只对约减轮有效，且攻击复杂度虽然比穷搜复杂度低，但是也超出了普通计算机的计算能力。近年来出现了可以对某些算法全轮攻击的攻击方法，如不变子空间 (Invariant Subspace) 攻击方法，这种方法不受轮数的限

制，且能在实际复杂度内将密码算法与随机置换区分开。随着对量子计算机研究的发展，对称密码的安全性强度将减半，这对算法的设计和分析都是挑战。

附录 A　KECCAK 碰撞攻击结果

A.1　KECCAK S 盒的 2-维可线性化仿射子空间

如表 A.1 所列，KECCAK S 盒共有 80 个 2-维可线性化仿射子空间。

表 A.1: KECCAK S 盒的 2-维可线性化仿射子空间

{0, 1, 4, 5}	{2, 3, 6, 7}	{0, 1, 8, 9}	{4, 5, 8, 9}	{0, 2, 8, A}
{1, 2, 9, A}	{0, 3, 8, B}	{1, 3, 9, B}	{2, 3, A, B}	{6, 7, A, B}
{0, 1, C, D}	{4, 5, C, D}	{8, 9, C, D}	{4, 6, C, E}	{5, 6, D, E}
{4, 7, C, F}	{5, 7, D, F}	{2, 3, E, F}	{6, 7, E, F}	{A, B, E, F}
{0, 2, 10, 12}	{8, A, 10, 12}	{1, 3, 11, 13}	{9, B, 11, 13}	{0, 4, 10, 14}
{1, 5, 10, 14}	{2, 4, 12, 14}	{0, 4, 11, 15}	{1, 5, 11, 15}	{3, 5, 13, 15}
{10, 11, 14, 15}	{0, 6, 10, 16}	{2, 6, 12, 16}	{3, 7, 12, 16}	{4, 6, 14, 16}
{C, E, 14, 16}	{1, 7, 11, 17}	{2, 6, 13, 17}	{3, 7, 13, 17}	{5, 7, 15, 17}
{D, F, 15, 17}	{12, 13, 16, 17}	{10, 11, 18, 19}	{14, 15, 18, 19}	{0, 2, 18, 1A}
{8, A, 18, 1A}	{10, 12, 18, 1A}	{11, 12, 19, 1A}	{10, 13, 18, 1B}	{1, 3, 19, 1B}
{9, B, 19, 1B}	{11, 13, 19, 1B}	{12, 13, 1A, 1B}	{16, 17, 1A, 1B}	{8, C, 18, 1C}
{9, D, 18, 1C}	{A, C, 1A, 1C}	{8, C, 19, 1D}	{9, D, 19, 1D}	{B, D, 1B, 1D}
{10, 11, 1C, 1D}	{14, 15, 1C, 1D}	{18, 19, 1C, 1D}	{8, E, 18, 1E}	{A, E, 1A, 1E}
{B, F, 1A, 1E}	{4, 6, 1C, 1E}	{C, E, 1C, 1E}	{14, 16, 1C, 1E}	{15, 16, 1D, 1E}
{9, F, 19, 1F}	{A, E, 1B, 1F}	{B, F, 1B, 1F}	{14, 17, 1C, 1F}	{5, 7, 1D, 1F}
{D, F, 1D, 1F}	{15, 17, 1D, 1F}	{12, 13, 1E, 1F}	{16, 17, 1E, 1F}	{1A, 1B, 1E, 1F}

A.2　KECCAK S 盒的差分分布表

表 A.2 给出了 KECCAK S 盒的差分分布表 [50]。

A.3　KECCAK 差分迹搜索结果

表 A.3 给出了 KECCAK 碰撞攻击所用差分迹核。

A.4　KECCAK 碰撞结果

表 A.4，A.5，A.6-A.8 给出了不同版本 5 轮 KECCAK 的碰撞消息。

表 A.2: KECCAK S 盒差分分布表

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
00	32	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
01	-	8	-	-	-	-	-	-	-	8	-	-	-	-	-	-	-	8	-	-	-	-	-	-	-	8	-	-	-	-	-	-
02	-	-	8	8	-	-	-	-	-	-	-	-	-	-	-	-	-	-	8	8	-	-	-	-	-	-	-	-	-	-	-	-
03	-	-	4	4	-	-	-	-	-	-	4	4	-	-	-	-	-	-	4	4	-	-	-	-	-	-	4	4	-	-	-	-
04	-	-	-	-	8	8	8	8	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
05	-	-	-	-	4	-	4	-	-	-	-	-	4	-	4	-	-	-	-	-	-	4	-	4	-	-	-	-	-	4	-	4
06	-	-	-	-	4	4	4	4	-	-	-	-	-	-	-	-	-	-	-	-	4	4	4	4	-	-	-	-	-	-	-	-
07	-	-	-	-	2	2	2	2	-	-	-	-	2	2	2	2	-	-	-	-	2	2	2	2	-	-	-	-	2	2	2	2
08	-	-	-	-	-	-	-	-	8	-	8	-	8	-	8	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
09	-	4	-	4	-	-	-	-	-	-	-	-	-	4	-	4	-	4	-	4	-	-	-	-	-	-	-	-	-	4	-	4
0A	-	-	-	-	-	-	-	-	4	-	-	4	4	-	-	4	-	-	-	-	-	-	-	-	4	-	-	4	4	-	-	4
0B	-	4	4	-	-	-	-	-	-	-	-	-	-	4	4	-	-	4	4	-	-	-	-	-	-	-	-	-	-	4	4	-
0C	-	-	-	-	-	-	-	-	4	4	4	4	4	4	4	4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
0D	-	-	-	-	4	-	4	-	4	-	4	-	-	-	-	-	-	-	-	-	-	4	-	4	-	4	-	4	-	-	-	-
0E	-	-	-	-	-	-	-	-	2	2	2	2	2	2	2	2	-	-	-	-	-	-	-	-	2	2	2	2	2	2	2	2
0F	-	-	-	-	2	2	2	2	2	2	2	2	-	-	-	-	-	-	-	-	2	2	2	2	2	2	2	2	-	-	-	-
10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	8	-	-	-	8	-	-	-	8	-	-	-	8	-	-	-
11	-	4	-	-	-	4	-	-	-	4	-	-	-	4	-	-	-	4	-	-	-	4	-	-	-	4	-	-	-	4	-	-
12	-	-	4	4	-	-	4	4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	4	4	-	-	4	4
13	-	-	2	2	-	-	2	2	-	-	2	2	-	-	2	2	-	-	2	2	-	-	2	2	-	-	2	2	-	-	2	2
14	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	4	4	-	-	-	-	4	4	4	4	-	-	-	-	4	4
15	-	4	-	-	-	-	-	4	-	4	-	-	-	-	-	4	4	-	-	-	-	-	4	-	4	-	-	-	-	-	4	-
16	-	-	4	4	4	4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	4	4	4	4	-	-
17	-	-	2	2	2	2	-	-	-	-	2	2	2	2	-	-	-	-	2	2	2	2	-	-	-	-	2	2	2	2	-	-
18	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	4	-	4	-	4	-	4	-	4	-	4	-	4	-	4	-
19	-	2	-	2	-	2	-	2	-	2	-	2	-	2	-	2	-	2	-	2	-	2	-	2	-	2	-	2	-	2	-	2
1A	-	-	-	-	-	-	-	-	4	-	-	4	4	-	-	4	4	-	-	4	4	-	-	4	-	-	-	-	-	-	-	-
1B	-	2	2	-	-	2	2	-	-	2	2	-	-	2	2	-	-	2	2	-	-	2	2	-	-	2	2	-	-	2	2	-
1C	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
1D	-	2	-	2	-	2	-	2	-	2	-	2	-	2	-	2	2	-	2	-	2	-	2	-	2	-	2	-	2	-	2	-
1E	-	-	-	-	-	-	-	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	-	-	-	-	-	-	-	-
1F	-	2	2	-	2	-	-	2	2	-	-	2	-	2	2	-	2	-	-	2	-	2	2	-	-	2	2	-	2	-	-	2

横坐标为输出差分，纵坐标为输入差分。

表 A.3: KECCAK 碰撞攻击所用差分迹核

用于 5-轮 SHAKE128 碰撞攻击的差分迹核	
β_2	----------|----------24|----------2-|----------|---------- ----------|----------4|----------|----------|---------4------ ----------|------4---------|----------2-|----------|---------- ----------|----------|----------|----------|---------- ---------4------|------4---------|---------4------|---------4------|---------4------
β_3	----------|----4----------|---1----------|----------|-----1---------- ----------|----4----------|----------|----------|---------- ----------8|----------|----------|----------|---------- ----------|----------|---1----------|----------|---------- ----------8|----------|----------|----------|-----1----------
β_4	----------|----------|----------|----------|2-4---------- ----------|----------8|----------4-|---------8------|---------- ----------|----------|----------|--8----------|---------2----- ----------4-8|----------4----|-1----------|8----------|---------- ----4----------|----------|----------4-----|----------|---1----------

用于 KECCAK[1440, 160, 5, 160] 碰撞攻击的差分迹核	
β_2	----------|----------|----------|----------|---------- --------1-------|----------8----|----------1--|--------1----1--|---------- --------1-------|----------|-1----------|-1----------|-1---------- ----------|----------8----|----------|----------|-1---------- ----------|----------|----------1--|----------1--|----------
β_3	----------|8---------|----------|----------|---------- ----------|----------|--------8-------|----------1|---------- ----------|----------|----------|----------1|---------- ----------|----------1|----------|----------|----------1 ----------|8---------|--------8-------|----------|----------
β_4	----------|----------|----------|----------|---------- ----------|----------|----------|----2----------|--------1------- ----------1|------2---------|---------2------|----------|---------- ----------|----------|----------4--|----------|---------- ----------|--8---------|----------|----------|----------2

用于 KECCAK[640, 160, 5, 160] 碰撞攻击的差分迹核	
β_2	----------|----------|----------|----------|---------- ----------1-|---2----------|----------|---2----------|----------1- ----------1|----------1|----------|---2----------|---------- ----------1-|----------1|----------|----------|---------- ----------|----------|----------|----------|----------
β_3	----------|----------|----------|----------|---------- ----------|----------|----------|----2----------|---------- ----------|----------|----------4--|----------|---------- ----------|-----1----------|----------4--|----------|---------- ----------|-----1----------|----------|----2----------|----------
β_4	----------|----------|--2----------|----------|---------- ----------|----------|----------2---|----------2-----|-----4---------- ----------|---8----------|----------|----------|---------- ----------|----------|----------|---------2------|------2--------- ----------|------1---------|----------|---8----------|-----4----------

表 A.4: KECCAK[1440, 160, 5, 160] 碰撞结果

M_1	C09C5501A913CC3C|7406D907E6569334|89182C870A0387A0|980A9D8F82C40A90|9306194AEBBC1C17 6D7DFE249ED35BB5|35C1981BFF84755C|37E7FA11AAD390EB|19485675C7530B8E|042893444D9EC364 6D317B9B40DE874C|E2EC2A3613678DDA|3939A7F72AC29BF6|4FABBC80AE5192EA|AB50ABCBCC7E5CC7 0F152006D01F65AC|AEC5B4B7EEC068E1|58E287388571520A|569ED102CB7D2EFA|4AC1C2A0645D5B2C 4C323DADBB2DAFC4|36F6BEEB558F2B22|0000000089F71BE8|0000000000000000|0000000000000000
M_2	D634EAE0EF26F002|90371C35BB5CFABC|7396C3D058D2F577|78CDF403D882B742|22ECA6BCBFC9501F 2352A9667EB05FCD|4CA3FD90EFB8A2D3|8DDFF276C0B60599|4B4CCD54AD6B2646|A490FAFA55BF4E37 234734EA58D9191D|3C580CA9664107ED|29E6AEB01815FB08|8FB33829BABDF8C2|48A21B6E764A7987 D9FA24DCB0331C80|9272D67CEF52F8E3|0C82810B4BE7307A|CF164B325F4DEEBE|BA41517B4D315C3C 99CD68FF39016FC4|AB018238479D9A8D|00000000E3233895|0000000000000000|0000000000000000
digest	A6E173DCDFC3E8EF|8242EAEA1EE736D5| E33875A0

表 A.5: 5-轮 SHAKE128 碰撞结果

M_1	0A3E44EBE62104A0|1E8617C352E80FBC|B69A38114369962F|1237F5EEA8045DAB|D4144AC64E22044C 1240A93D79FCCB2E|C8C63A830CBACFFC|B36B34C0E1719824|F94803ECC5586680|ACF133FE29839CAD CA5F88F260DEFAA7|972FE7E882A4AB03|D11344BE12431A54|814488EBAE68F93B|56D10CF0251FAED3 77A665FCC5F52D9F|D50EF69FAB128ACC|87F3F1816E740894|770D4D55489234B0|737134B1243F3A3D FE0E2AE7F23D8E40|0000000000000000|0000000000000000|0000000000000000|0000000000000000
M_2	0CDDD5D25A8BD7CA|F71D259EE445A4D8|EB84F177D51C9D45|0A70C1FC50024C24|7108096E3F024F63 3B8D1EEBC5E9150E|EADCC9FB19824E75|B8A97CB74697BAAD|5988E2CD64063AD1|FB55123185E2E4A4 E74FF74033CA1486|915F016B41BDAF6B|145441AAC9EFA342|D9A609CF15E6C626|5609C4F58F5DEE0A AB4E178C43BA8687|3774B01D78F2ABE4|AA35E3D371664594|A26EAD50F73069A7|DE4E25F8A0F8E928 FE431BE34F8371D8|0000000000000000|0000000000000000|0000000000000000|0000000000000000
digest	9D2E953AD7C6A939|326F59A68A6016EF|A71EAFEE371700D7|3C463D5D098D9B76

表 A.6: KECCAK[640, 160, 5, 160] 碰撞结果 1

M_1	297DB73F|CE5FB46D|63EFD5AB|AB75DBB2|020119E7 06927773|A645A6A4|68E6E3F8|15282462|633AAB83 96C7A5FB|5E4CBEB5|92614C96|DD9647DA|D4B0094F 4C68376F|D3B63751|6286AB56|DE577A52|9003EA0F 00000000|00000000|00000000|00000000|00000000	M_2	5B150BCB|C0F3F2FC|5907B5A5|22736DC3|914CF0C5 87477D63|A675A649|8BBEA96F|52EB8AE3|19402D41 D9FB4CC3|669FD630|D8C9FC71|57558554|0662F64A 64B4B5C5|7F12BF56|2BEADBF0|F6207B10|F2FD9787 00000000|00000000|00000000|00000000|00000000
digest	F90B5ABA|7430682D|85668C62|66E1B0AD|B052AC35		

表 A.7: KECCAK[640, 160, 5, 160] 碰撞结果 2

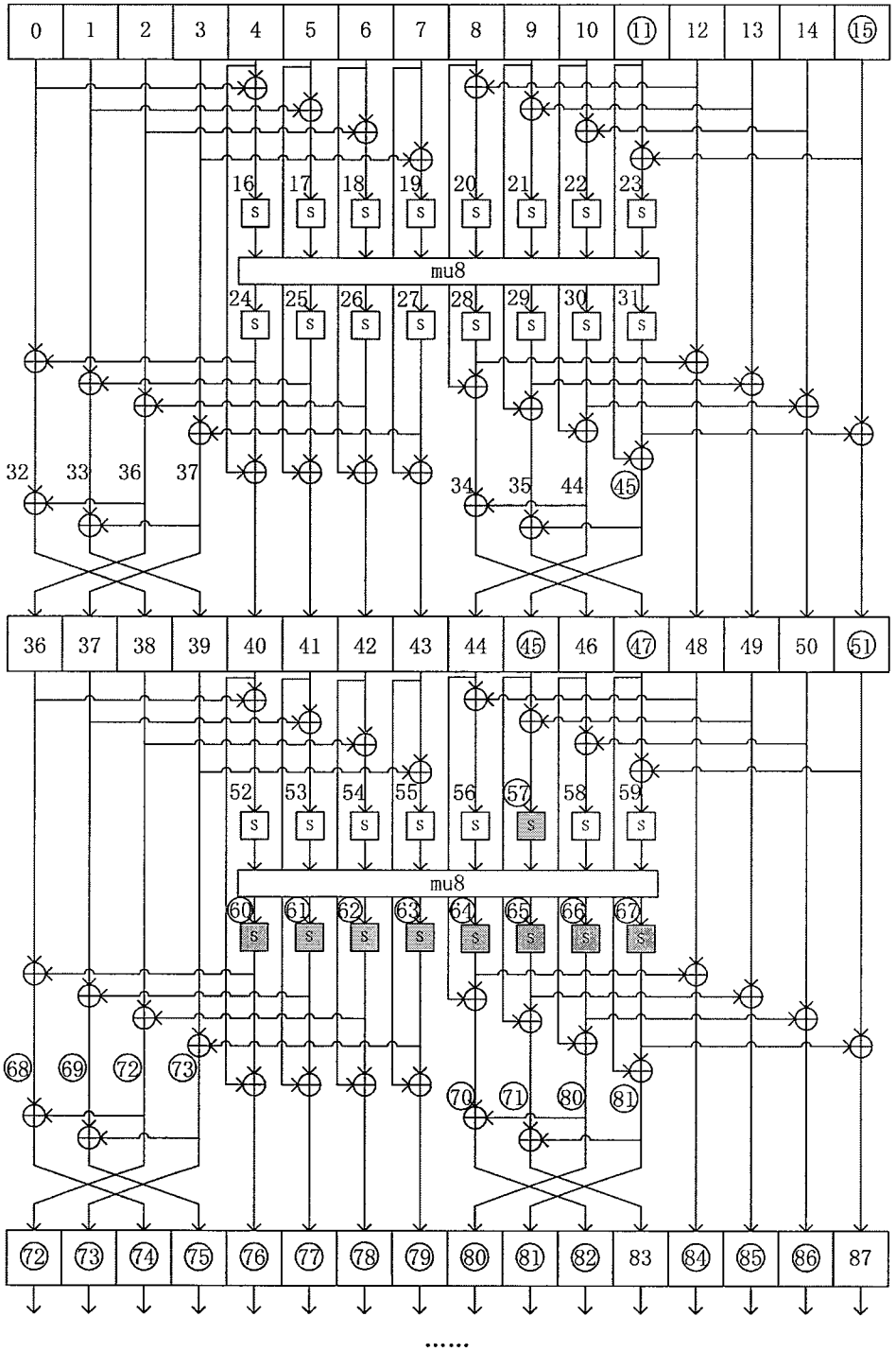
M_1	1F251BDB|F4C3F23F|ABAFE5A5|62F245D1|90C9E900 E757FD6B|8695EF8A|E836BBE7|7C2AAA21|4545A880 BC6BCC59|EEFE86B5|D241C27A|1E94E556|4257E1CA 00A435C7|FF63F7D3|22A2CB51|DEC53302|B6D8BE13 00000000|00000000|00000000|00000000|00000000	M_2	6D4DA72F|FA6FB4AE|914785AB|EBF4F3A0|03840022 6682F77B|86A5EF67|0B6EF170|3BE904A0|3F3F2E42 F3572561|D62DEE30|98E9729D|945727D8|90851ECF 2878B76D|53C77FD4|6BCEBBF7|F6B23240|D426C39B 00000000|00000000|00000000|00000000|00000000
digest	C3A0AC92|432FD88D|185C7A87|A6071DFC|7B3888E7		

表 A.8: KECCAK[640, 160, 5, 160] 碰撞结果 3

M_1	0318BF1B|0CD7322A|30CBD72F|C13773E1|9660142E 0F5E7522|E69527AD|49FAB2EC|75EB9CE0|4B4C6C4C 8B93E419|869A96B4|9165CEB1|FCD4AD1A|447E9E47 54AC979D|1B57BE12|20CECB52|B9215347|94C1FBD8 00000000|00000000|00000000|00000000|00000000	M_2	717003EF|027B74BB| 0A23B721|4831C590|052DFD0C 8E8B7F32|E6A52740|AAA2F87B|32283261|3136EA8E C4AF0D21|BE49FE31|DBCD7E56|76176F94|96AC6142 7C701537|B7F33615|69A2BBF4|91565205|F63F8650 00000000|00000000|00000000|00000000|00000000
digest	0C2AA9EE|CC322FE1|BEB00FCA|F92B6839|3CE7F855		

附录 B　FOX128 无效差分迹

如图 B.1 所示，数字是对字级变量的编号，圆圈表示该位置的差分非 0，无圆圈表示该位置差分为 0。令 x_i 表示编号为 $i(i=0,1,2,\cdots)$ 的字级差分值，$\underline{x_i}$ 表示差分 x_i 经 S 盒后的输出差分值，带阴影的 S 盒是活跃 S 盒。在该差分迹中存在矛盾，一方面有 $x_{23}=x_{11}\oplus x_{15}\neq 0$ 和 $x_{47}=x_{45}=x_{11}, x_{51}=x_{15}$，得到 $x_{47}\neq x_{51}$；另一方面有 $x_{59}=x_{47}\oplus x_{51}=0$，得到矛盾，因此这是一条无效差分。



圆圈位置差分非 0，无圆圈位置差分为 0，$\underline{x_i}$ 表示差分 x_i 经 S 盒的输出差分值，阴影 S 盒活跃。一方面有 $x_{23} = x_{11} \oplus x_{15} \neq 0$ 和 $x_{47} = x_{45} = x_{11}, x_{51} = x_{15}$，另一方面有 $x_{59} = x_{47} \oplus x_{51} = 0$，得到矛盾。

图 B.1: FOX128 无效差分迹

附录 C SIMON 和 Simeck 差分迹

C.1 解密方向差分方程的建立与求解

解密方向上的差分方程为 $\Delta X_j^i = 0$ 或 $\Delta X_j^i = 1, j \in [0, n-1]$，其中

$$\begin{aligned}\Delta X_j^i =& \Delta X_{(j+b)\%n}^{i+1} \wedge X_{(j+a)\%n}^{i+1} \oplus \Delta X_{(j+a)\%n}^{i+1} \wedge X_{(j+b)\%n}^{i+1} \\ & \oplus \Delta X_{j+b}^{i+1} \wedge \Delta X_{(j+a)\%n}^{i+1} \oplus \Delta X_{(j+c)\%n}^{i+1} \oplus \Delta X_j^{i+2},\end{aligned} \tag{C.1}$$

并有

$$\begin{aligned}X_{(j+a)\%n}^{i+1} =& X_{(j+a+b)\%n}^{i+2} \wedge X_{(j+a+a)\%n}^{i+2} \oplus X_{(j+a+c)\%n}^{i+2} \\ & \oplus X_{(j+a)\%n}^{i+3} \oplus K_{(j+a)\%n}^{i+1}, \\ X_{(j+b)\%n}^{i+1} =& X_{(j+b+b)\%n}^{i+2} \wedge X_{(j+b+a)\%n}^{i+2} \oplus X_{(j+b+c)\%n}^{i+2} \\ & \oplus X_{(j+b)\%n}^{i+3} \oplus K_{(j+b)\%n}^{i+1}.\end{aligned} \tag{C.2}$$

算法 3 给出了影响比特 X_j^i 的子密钥比特和与之线性相关的子密钥比特。

C.2 Simeck 扩展差分迹

22 轮 Simeck32/64、28 轮 Simeck48/96 和 35 轮 Simeck64/128 的扩展差分迹在表 C.1, C.2 和 C.3 中给出。

C.3 SIMON 扩展差分迹

22 轮 SIMON32、24 轮 SIMON48 和 29、30 轮 SIMON64 的扩展差分迹在表 C.4，C.5，C.6 和 C.7 中给出。

Algorithm 3 X_j^i 相关子密钥比特 (解密方向)

输入: 轮数 i，比特位置 j
输出: $[Influent\_keys, Linear\_keys]$ ▷ [有影响子密钥，线性相关子密钥]
function RELATEDKEYS(i, j)
  $Influent\_keys$= [ ], $Linear\_keys$=[ ]
  if $i = r_0 + R + r_1$ **then**
    return $[Influent\_keys, Linear\_keys]$
  else
    if $j \geq n$ **then**
      return RELATEDKEYS$(i+1, j\%n)$
    else
      $[I_0, L_0]$=RELATEDKEYS$(i, (j+a)\%n+n)$
      $[I_1, L_1]$=RELATEDKEYS$(i, (j+b)\%n+n)$
      $[I_2, L_2]$=RELATEDKEYS$(i, (j+c)\%n+n)$
      $[I_3, L_3]$=RELATEDKEYS$(i+1, j+n)$
      $Linear_keys$=$L_2 \cup L_3 \cup K_j^i$
      $Influent\_keys = I_0 \cup I_1 \cup I_2 \cup I_3 \cup K_j^i$
      return $[Influent\_keys, Linear\_keys]$
    end if
  end if
end function

表 C.1: 22-轮 Simeck32 扩展差分迹

轮数	每轮输入差分
0	$0,0,0,*,*,0,0,*,*,*,0,1,*,*,*,*,0,0,*,*,*,0,*,*,*,*,*,*,*,*,*,*$
1	$0,0,\mathbf{0},\mathbf{0},*,0,\mathbf{0},\mathbf{0},*,*,\mathbf{0},\mathbf{0},\mathbf{1},*,*,\mathbf{0},0,0,0,*,*,0,0,*,*,*,0,1,*,*,*,*$
2	$0,0,0,\mathbf{0},\mathbf{0},0,0,\mathbf{0},\mathbf{0},*,0,0,\mathbf{0},\mathbf{1},*,\mathbf{0},0,0,0,0,*,0,0,0,*,*,0,0,1,*,*,0$
3	$0,0,0,0,\mathbf{0},0,0,0,\mathbf{0},\mathbf{0},0,0,0,\mathbf{0},\mathbf{1},0,0,0,0,0,0,0,0,0,*,0,0,0,1,*,0$
4	$0,0,0,0,0,0,0,0,0,\mathbf{0},0,0,0,0,\mathbf{0},0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0$
4→17	13-轮差分区分器
17	$0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,\mathbf{0},0,0,0,0,\mathbf{0},0$
18	$0,0,0,0,0,0,0,0,0,*,0,0,0,1,*,0,0,0,0,0,\mathbf{0},0,0,0,\mathbf{0},\mathbf{0},0,0,0,\mathbf{0},\mathbf{1},0$
19	$0,0,0,0,*,0,0,0,*,*,0,0,1,*,*,0,0,0,0,\mathbf{0},\mathbf{0},0,0,\mathbf{0},\mathbf{0},*,0,0,\mathbf{0},\mathbf{1},*,\mathbf{0}$
20	$0,0,0,*,*,0,0,*,*,*,0,1,*,*,*,*,0,0,\mathbf{0},\mathbf{0},*,0,\mathbf{0},\mathbf{0},*,*,\mathbf{0},\mathbf{0},\mathbf{1},*,*,\mathbf{0}$
21	$0,0,*,*,*,0,*,*,*,*,*,*,*,*,*,*,0,\mathbf{0},\mathbf{0},*,*,\mathbf{0},\mathbf{0},*,*,*,\mathbf{0},\mathbf{1},*,*,*,*$
22	$0,*,*,*,*,*,*,*,*,*,*,*,*,*,*,*,*,\mathbf{0},\mathbf{0},*,*,*,\mathbf{0},*,*,*,*,*,*,*,*,*,*$

表 C.2: 28-轮 Simeck48 扩展差分迹

轮数	每轮输入差分
0	***000000***0**********0***0**********
1	***00000000000***0****1****000000***0**********
2	***000000000000000***01***00000000000***0****1*
3	111000000000000000000000***0000000000000000***01
4	010000000000000000000001110000000000000000000000
4→24	20-轮差分区分器
24	010000000000000000000000100000000000000000000000
25	1*1000000000000000*000010000000000000000000000
26	***000000000000*000***011*1000000000000000000*000
27	***0000000*000***0****1****000000000000*000***01
28	***00*000***0**********0000000*000***0****1*

表 C.3: 35-轮 Simeck64 扩展差分迹

轮数	每轮输入差分
0	**********0000000*000**00***0**********00*000**00***0*******
1	*0****1***000000000000*000**00**********0000000*000**00***0***
2	*00***01**0000000000000000*000**0****1***000000000000*000**00**
3	*000**001*00000000000000000000000*00***01**0000000000000000*000*
4	000001000100000000000000000000000*000**001*0000000000000000000000
5	000000000000000000000000000000000000000100010000000000000000000000
5→31	26-轮差分区分器
31	000010001000000000000000000000000000000100000000000000000000000
32	000**001*1000000000000000000000*000010001000000000000000000000000
33	00***01***0000000000000000*000**000**001*1000000000000000000000*
34	0****1****00000000000*000**00***00***01***0000000000000000*000**
35	**********000000*000**00***0****0****1****00000000000*000**00***

表 C.4: 22-轮 SIMON32 扩展差分迹

轮数	每轮输入差分
0	00**00001**0*000***0*01*****0000
1	0000*000001*000000**00001**0*000
2	00000000000010000000*000001*0000
3	00000000000000000000000000001000
3→17	14-轮差分区分器
17	00001000000000000000000000000000
18	001*00000000*0000001000000000000
19	1**0*00000**0000001*00000000*000
20	****0000***0*01*1**0*00000**0000
21	***0*0******1*******0000***0*01*
22	********************0*0******1***

表 C.5: 24-轮 SIMON48 扩展差分迹

轮数	每轮输入差分
0	00*0***01*1***11*00**0*0****************0******0
1	000000*000*01**00*001*0000*0***01*1***11*00**0*0
2	000000000000001000100010000000*000*01**00*001*00
3	000000000000000010000000000000000000001000100010
3→20	17-轮差分区分器
20	000000000000001000100010000000000000000010000000
21	000000*000*01**00*001*000000000000000001000100010
22	00*0***01*1***11*00**0*0000000*000*01**00*001*00
23	****************0******000*0***01*1***11*00**0*0
24	***************1**********************0******0

表 C.6: 29-轮 SIMON64 扩展差分迹

轮数	每轮输入差分
0	*0***1*0000*000*0**00*****0**********0*0**00*****0**********
1	**011*0100000000000*000*0**001*1*0***1*0000*000*0**00*****0*****
2	01*000*000000000000000000000*000***011*0100000000000*000*0**001*1
3	000100010000000000000000000000001*000*000000000000000000000*000*
4	000001000000000000000000000000000000010001000000000000000000000000
4→25	21-轮差分区分器
25	000100010000000000000000000000000000010000000000000000000000000000
26	01*000*000000000000000000000*000*000100010000000000000000000000000
27	**011*0100000000000*000*0**001*101*000*000000000000000000000*000*
28	*0***1*0000*000*0**00*****0*******011*0100000000000*000*0**001*1
29	******0*0**00*****0**************0***1*0000*000*0**00*****0*****

表 C.7: 30-轮 SIMON64 扩展差分迹

轮数	每轮输入差分
0	****0******************010**000**********************0****0***
1	**000****000****0******0***1*0000000****0******************010**000
2	**00000000000****000*1**011***010000000000**000****0******0***1*0000000
3	000000000000000000001100010000000000000000000**000*1**011*01000000
4	000000000000000000000100010000000000000000000000000011000100000000
4→26	22-轮差分区分器
26	000000000000000000000100010000000000000000000000000000000100000000
27	0000000000000*000*01*000*000000000000000000000000000010001000000
28	00000*000*0**001*1**011*010000000000000000000*000*01*000*0000000
29	0*0**00*****0******0***1*0000*0000000*000*0**001*1**011*01000000
30	****0******************0*0**00*0*0**00*****0******0***1*0000*00